HOCHSCHULE
MITTWEIDA
UNIVERSITY OF
APPLIED SCIENCES

BACHELORARBEIT

Herr
Phillip Träber

Konzeption und prototypische Entwicklung eines Debuggers zum Betrachten und Manipulieren des Ash-Frameworks zur Laufzeit in ActionScript 3

2015

Fakultät: **Mathematik/Naturwissenschaften/Informatik**

BACHELORARBEIT

Konzeption und prototypische Entwicklung eines Debuggers zum Betrachten und Manipulieren des Ash-Frameworks zur Laufzeit in ActionScript 3

Autor:
Phillip Träber

Studiengang:
Medieninformatik und interaktives Entertainment

Seminargruppe:
MI11w1-B

Erstprüfer:
Prof. Dr.-Ing. Wilfried Schubert

Zweitprüfer:
M.Sc. Dipl.-Inf. (FH) Knut Altroggen

Mittweida, Januar 2015

Bibliografische Angaben

Träber, Phillip: Konzeption und prototypische Entwicklung eines Debuggers zum Betrachten und Manipulieren des Ash-Frameworks zur Laufzeit in ActionScript 3, 51 Seiten, 26 Abbildungen, Hochschule Mittweida (FH), University of Applied Sciences, Fakultät Mathematik/Naturwissenschaften/Informatik

Bachelorarbeit, 2015

Referat

Die Bachelorarbeit beschäftigt sich mit Möglichkeiten, Objekte zur Laufzeit zu manipulieren. Dabei werden Verfahren zur Analyse von Klassen und deren Werte, sowie die Versendung an ein externes Tool behandelt. Zudem werden diese Daten über eine Benutzeroberfläche präsentiert. Dabei spezialisiert die Arbeit sich auf das Ash-Framework und wird mithilfe von ActionScript 3 realisiert. Abschließend wird das Framework an einem Praxisbeispiel angewandt.

I. Inhaltsverzeichnis

II. Abbildungsverzeichnis

III. Vorwort

Als Student in der Fachrichtung „**Medieninformatik und interaktives Entertainment**“ wird ein Praxissemester für das Erreichen des Abschlusses vorausgesetzt. In diesem Zeitraum sollte sich ein Betrieb gesucht werden, in dem als Praktikant ersten Einblicke in das Arbeitsleben gezeigt werden.

Dies ist auch sehr gut gelungen. Dazu hatte ich Gelegenheit, während meiner Praktikumszeit und der Anfertigung meines Bachelorthemas für ein knappes Jahr im Hamburger Computerspieleunternehmen Goodgame Studios zu arbeiten. Die meisten Projekte, darunter auch die, an denen ich gearbeitet habe, wurden mit Flash und ActionScript 3 realisiert. Da die Teammitglieder oft intern von Projekt zu Projekt wechselten und die Firma stetig neue Mitarbeiter einstellt, wurden simple Fehler im Spiel schnell zur Jagt nach der Nadel im Heuhaufen. Viele Klassen wurden von Programmierern erstellt, welche längst nicht mehr anwesend waren. Und so war es notwendig, sich erst in vorhandene Klassen einzuarbeiten und deren Abhängigkeit zu überblicken.

So kam ich auf die Idee für meine Bachelorarbeit. Die Entwicklung eines funktionierenden Debuggers, welcher mit dem Ash-Framework arbeiten kann.

Ich möchte mich noch bei meiner Firma dafür bedanken, dass sie mir die Zeit und notwendigen Mittel zur Verfügung gestellt haben, welche ich zur Fertigstellung der Bachelorarbeit benötigte.

1 Einleitung

Im folgenden Kapitel wird erklärt, wie es zu dieser Arbeit kam und was das Thema beinhaltet.

1.1 Motivation

Die meisten IDE's[1] besitzen einen integrierten Debugger. Je nach Editor und verwendeter Programmiersprache ist dieser relativ mächtig. Hiermit können zur Laufzeit alle aktuellen Werte von Variablen betrachtet und verändert werden, während die Anwendung in einer Art Ruhemodus verharrt. Doch diese Art von Debugger ist nicht gut für das Arbeiten mit einem Entity-System-Framework, da dieses eine andere Struktur und Logik zur Ausführung des Quellcodes besitzt.

Bei der Entwicklung von Computerspielen ist es keine Seltenheit, dass sich während des kompletten Zeitraums die Inhalte und Anforderungen wiederholt ändern. Gründe hierfür sind geplante Mechaniken, welche nicht wie erhofft funktionieren oder durch bessere Einfälle ersetzt werden. All diese Änderungen haben Auswirkungen auf den Quellcode. Wurde dieser anfänglich zum Beispiel auf ein Rennspiel ausgelegt und später wird entschieden, dass der Spieler auch aus dem Fahrzeug aussteigen und Häuser betreten kann, so muss die vorhandene Struktur an die neuen Anforderungen angepasst werden, auch wenn das Grundgerüst nicht darauf ausgelegt ist.

Diese Änderungen führen dazu, dass immer mehr Klassen hinzugefügt werden, welche nicht einhundert prozentig aufeinander abgestimmt sind. Im Laufe des Entwicklungsprozesses entstehen so etliche Systeme und es ist schwer, den Überblick zu behalten, zu welchem Zeitpunkt wie viele Systeme aktiv sind. Meist fällt ein Problem erst auf, wenn das Spiel nicht mehr flüssig läuft oder Fehler auftreten.

Ein Debugger, der auf solch ein Framework zugeschnitten ist, könnte genutzt werden, um die Probleme schneller zu beheben oder sogar vorzeitig aufzudecken.

Doch nicht nur Programmierer können Gebrauch von diesem Tool machen. Sogenannte **Gamebalancer** kümmern sich um das Gleichgewicht in jedem Spiel. Sie achten darauf, dass dieses für den Spieler weder zu schwer, noch zu einfach ist. Dies erreicht er mit zahlreichen kleinen Anpassungen der Werte von einer Vielzahl von Spielobjekten. So bestimmt ein Gamebalancer zum Beispiel die Sprunghöhe einer Figur oder die Stärke des Gegners. Um den perfekten Wert für eine bestimmte Sache zu finden, muss er mehrmals das Spiel testen und immer wieder die Werte anpassen. Dies dauert aber

[1] IDE - (integrated development environment) Integrierte Entwicklungsumgebung

in der Regel sehr lange, da zum einen das Spiel erneut gestartet werden muss und manchmal eine dritte Person für das Ändern der Werte im Quellcode notwendig ist.

Mithilfe eines externen Debuggers könnte der Gamebalancer jedoch diese Änderung während des Spielens ändern und die Auswirkungen sofort testen. Ist er mit dem Ergebnis zufrieden, so kann er die Änderungen final ins Spiel überragen.

1.2 Ziel der Arbeit

Ziel der Arbeit ist es, einen lauffähigen Debugger zu konzipieren. Dieser ist für Anwendungen bestimmt, welche mit ActionScript 3 realisiert sind und das Ash-Framework von Richard Lord nutzen. Im Mittelpunkt stehen vor allem das Entity-System und entsprechende Systeme.

Da das Framework wohl hauptsächlich in Spielentwicklungen Einsatz findet, wird auch der Debugger mit Bedacht auf dieser Tatsache konzipiert. Das bedeutet, dass er das Spiel so wenig wie möglich beeinflussen darf. Das beinhaltet sowohl Logik und Ablauf als auch die Performance.

Zum Thema gehört unter anderem die Kommunikation zwischen dem Debugger und der Applikation. Hier werden Lösungsansätze erläutert, um Daten zwischen beiden Anwendungen hin und her zu senden. Dies beinhaltet sowohl den Verbindungsaufbau als auch das Analysieren und Verpacken der notwendigen Klassen.

Der Debugger soll Informationen der laufenden Applikation darbieten. Pflichtkriterien hierbei sind eine Ansicht aller aktiven Systeme, sowie eine Baumstruktur ähnliche Darstellung der Enities inklusive der zugehörigen Komponenten und Werte. Systeme sowie Entities und Komponenten können mit dem Programm aktiviert bzw. deaktiviert werden um so auf das Spiel Einfluss zu nehmen.

Für eine zeitunabhängige und ruhige Betrachtung der Werte soll die Möglichkeit bestehen, die Applikation zu Pausieren. Vielleicht ist auch über eine Art Breakpointsystem nachzudenken (siehe Kapitel Debugger).

Des Weiteren wird untersucht, in wie fern die Möglichkeit besteht ein Abbild des aktuellen Zustandes der Engine zu erschaffen. Zu einem späteren Zeitpunkt besteht so die Möglichkeit, diesen Zustand wiederherzustellen. Auch eine Realisierbarkeit zur Erstellung von neuen Entities wird überprüft.

2 Grundlagen und Begriffe

Folgendes Kapitel beschäftigt sich mit einigen Grundlagen, welche nötig sind, um die kommenden Problematiken und Thesen vollständig zu verstehen. Dabei wird sowohl auf verwendete Programme, sowie genutzte Klassen eingegangen.

2.1 GameEngine

Eine GameEngine stellt den Part eines Spieles dar, welcher die Grundlegenden Software Komponenten enthält und diese vom eigentlichen Spiel aus kapselt. Diese Komponenten können für das Rendern von 3D-Grafiken, das Abspielen von Sounds und der Berechnung der Kollisionen verantwortlich sein. Auf der anderen Seite stehen Spiel relevante Inhalte wie Art-Assets, Maps und Spiellogik. Die Grenze zwischen Spiel und Engine ist jedoch nicht immer klar.

Ziel einer **GameEngine** ist es, sie so zu konstruieren, dass sie für weitere Projekte genutzt werden kann, ohne große Anpassungen vorzunehmen. So wird den Entwicklern ermöglicht, schnell einen weiteren Teil einer Spielreihe zu produzieren. Man kann die einzelnen Engines nach ihrer Wiederverwendbarkeit einordnen. In Abbildung 2.1 stehen Links die Engines, welche nur für das eine Projekt genutzt werden können. Für nachfolgende Spiele, welche sich nur leicht ändern, muss eine komplett neue Struktur erarbeitet werden. Ganz rechts befindet sich die “perfekte„ universal Lösung. Diese Engine kann für jede mögliche Form eines Spiels verwendet werden. Eine solche Engine gibt es in der Realität nicht. Die meisten Engines sind auf ein gewisses Genre ausgelegt. Zum Beispiel wird die CryEngine voranging für Shooter genutzt. Hiermit ein Aufbauspiel zu realisieren, wird eher schwierig.

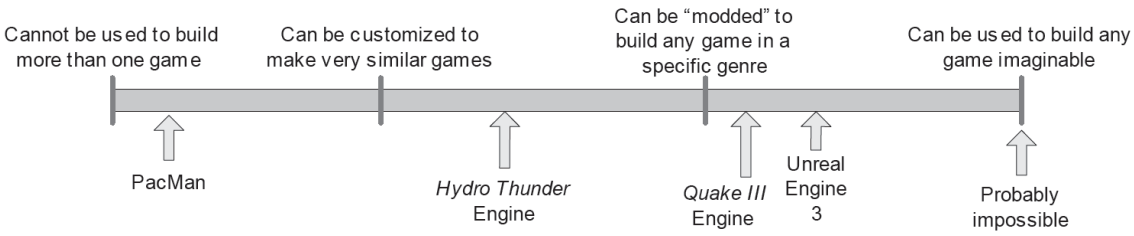

Abbildung 2.1: GameEngine - Quelle: Gregory Jason, Game Engine Architecture 2009, S.12

Eine Engine auf viele möglich Genres auszulegen, geht meist auf Kosten der Performance. Eine Engine kann nur optimiert werden, wenn dem Nutzer gewisse Regeln vorgeschrieben werden. Sei dies das Genre oder die verwendete Zielplattform. Doch durch die ständige Weiterentwicklung der Hardware, wird dieser Fakt wohl immer unbedeutender. Entwickler können sich professionelle GameEngines lizenzieren lassen um diese für ihre Projekte zu nutzen.

2.2 GameLoop

Kaum eine GameEngine kommt ohne GameLoop aus. In älteren Spielen, welche rein Textbasiert waren, sogenannte TextAdventure, lief das Spiel nach folgendem Schema ab. Code wird ausgeführt –> Text wird angezeigt –> warten auf Nutzereingabe <–>. Während auf Eingaben vom Nutzer gewartet wird, ist es nicht möglich andere Codeteile

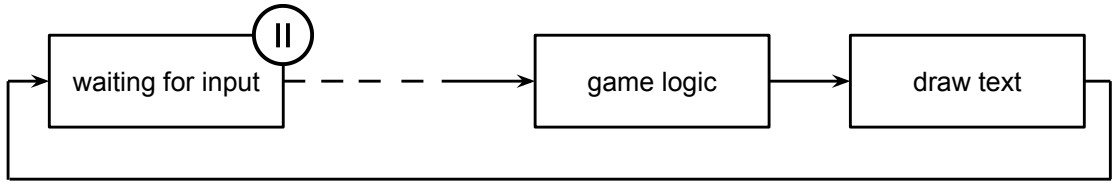

Abbildung 2.2: old game sequence

auszuführen. Das Spiel befindet sich in einer Art Pausenmodus. Dies war früher bei solch einem Spielkonzept auch kein Problem. Doch heutzutage passieren viele Dinge auf dem Bildschirm, auch wenn der Nutzer keine Eingabe tätigt. Effekte, Animationen und Sound sind ständig präsent. Auch die Gegner reagieren und Blätter an Bäumen wehen im Wind. Um dies zu ermöglichen, ist ein GameLoop nötig. Dieser sorgt dafür, dass in regelmäßigen Abständen Objekte aktualisiert werden und auf Nutzereingaben reagiert wird, ohne dass das Spiel in kompletten Ruhezustand verfällt.

Ein solcher GameLoop könnte wie in Abbildung 2.3 aussehen. Meist werden in einer

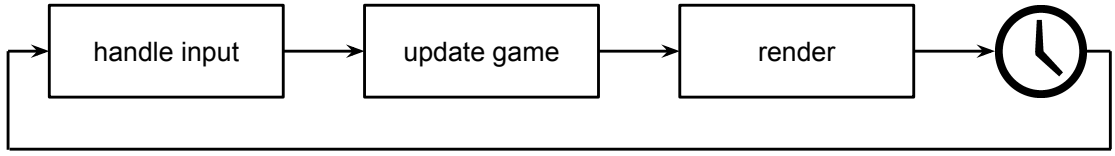

Abbildung 2.3: GameLoop

While-Schleife drei Zustände durchlaufen. Zunächst wird überprüft, ob irgendwelche Eingabesignale getätigt wurden. Das kann ein Mausklick sein, jedoch auch das Verschieben oder Schließen des Fensters. Danach wird jegliche Spiellogik ausgeführt und Objekte aktualisiert. Dazu gehört auch die Berechnung der Physik und der Kollision. Zum Schluss wird der aktuelle Zustand auf den Bildschirm gezeichnet. Nun wiederholt sich dieser Vorgang bis die Schleife abgebrochen wird.

Zu beachten ist bei diesem Ansatz, dass die Schleifendurchläufe so oft und schnell wie möglich abgearbeitet werden. Daraus folgt, dass das Spiel auf unterschiedlichen Rechnern unterschiedlich schnell läuft. Dies ist abhängig von der Leistungsfähigkeit der verwendeten Hardware.

Um dies zu umgehen, wird beispielsweise versucht eine stabile FrameRate zu erzwin-

gen, indem am Ende der Schleife die restliche Zeit gewartet wird, bis ein neuer Durchlauf starten soll. Eine weitere Möglichkeit ist die Übergabe der verstrichenen Zeit seit dem letzten Durchlauf an die Updatefunktion. So kann ein Spiel mit unterschiedlicher Hardware gespielt werden ohne das das Gameplay beeinflusst wird. Lediglich Abstriche bei der flüssigen Darstellung müssen bei älteren Geräten gemacht werden. Viele PhysicEngines benötigen eine konstante Aufrufzeit. Kann diese nicht sichergestellt werden, so verhalten sich Objekte nicht wie erwartet. Das können zum Beispiel Autos sein, welche plötzlich hochspringen oder Kugeln, die bei hoher Geschwindigkeit durch Objekte fliegen, anstatt sie zu treffen.

Bei der Verwendung externer GameEngines gibt es meist nicht die Möglichkeit, auf den Mainloop Einfluss zu nehmen. Es ist jedoch stets von Vorteil, zu wissen wie dieser intern aufgebaut ist. Dadurch können unnötige Aufrufe minimiert werden.

2.3 Entwicklungsprozess

Nicht nur die Spielinhalte haben sich über die Zeit verändert, sondern auch der Entwicklungsprozess. So war früher oft nur ein einziger Entwickler an einem Projekt beteiligt. Dieser kümmerte sich teilweise um die Programmierung, den Sound und die Grafiken. Dementsprechend klein und überschaubar waren diese Projekte.

Heutzutage arbeiten hunderte an Entwicklern gleichzeitig an einem Projekt, welches sich über Jahre erstreckt. In diesem Zeitraum kommt es nicht selten vor, dass sich der Personalstamm ändert. Neue Mitarbeiter müssen sich in vorhandenen Code einarbeiten und diesen verstehen. In Abbildung 2.4 ist ein kompletter Entwicklungsprozess dargestellt. Startpunkt jeder Spielumsetzung ist eine Idee. Diese wird meist zuerst als Pro-

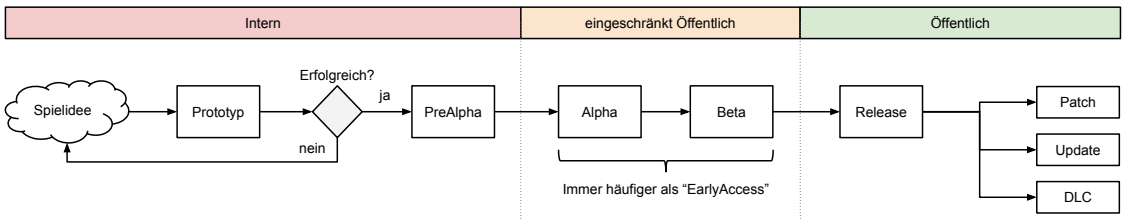

Abbildung 2.4: Entwicklungsprozess

totyp auf Tauglichkeit getestet. Das Ziel eines Prototyps ist es, schnellstmöglich Spielmechaniken umzusetzen. Dadurch wird auf geordnete Strukturen verzichtet. Das Spiel muss einfach laufen, auch wenn die komplette Spiellogik in der Main-Klasse steht.

Konnte die Umsetzbarkeit der vorangegangen Ideen bestätigt werden, so wird das Projekt neu aufgesetzt und entsprechende Design-Patterns angewandt. In dieser Phase ist es besonders wichtig, eine gute Code-Basis zu schaffen um später viele Überarbeitungen zu vermeiden. Sobald erste Abschnitte und Features getestet werden können,

befindet sich das Spiel in der PreAlpha. In größeren Firmen gibt es eine eigene Abteilung, welche für das Testen der implementierten Teile zuständig ist. Wurden Fehler (Bugs) oder falsches Verhalten auf ein Ereignis festgestellt, so wandert diese Aufgabe zurück zum Entwickler und muss von ihm angepasst werden. Dies geschieht so lange bis kein Fehlverhalten mehr festgestellt werden kann.

Befindet sich ein Spiel in fortgeschrittener Phase, so spricht man von einer **Alpha-** oder **Beta-Phase**. Wobei eine Beta schon sehr dem Endprodukt ähnelt und ab diesem Moment keine weiteren Features hinzukommen sollten. Diese beiden Phasen werden von Spiel zu Spiel auch an die Öffentlichkeit getragen. Bei sogenannten Alpha-/Beta-Tests können Nutzer meist kostenfrei Einblick ins Spiel erhalten und die Entwickler über Fehler informieren. Oft ist eine Beta anfangs nur für eine begrenzte Anzahl verfügbar. Eine sogenannte Closed-Beta. Später ist diese auch für den Rest verfügbar. Immer mehr Anklang findet das EarlyAccess-System. Hierbei wird bereits die Alpha verkauft. Kunden können so das Spiel deutlich eher spielen, müssen dafür jedoch noch mit vielen Fehlern rechnen. Dieses Vorgehen wird problematisch, wenn das Spiel nie fertig gestellt wird. Kunden bleiben als Folge auf dem unfertigen Produkt sitzen. Dieses Szenario nennt man eine perpetual Beta.

Wurde ein Spiel veröffentlicht, so hört die Entwicklung nicht unmittelbar danach auf. Oft werden die nächsten Wochen abgewartet und Fehler mithilfe von Patches gefixt. Das sind kleine Updates welche von den Spielern heruntergeladen werden können um das Spielgefühl zu verbessern. Manche Projekte laufen auch direkt in die zweite Phase und kümmern sich um die Erstellung eines Nachfolgers. Dieser kann auf der selben Codebasis aufbauen und somit beträchtliche Entwicklungszeit sparen.

Gerade bei Browsergames muss nach dem Release weiterhin entwickelt werden, um dem Spieler ein erweitertes Spielerlebnis zu ermöglichen. Nur hierdurch können Kunden dauerhaft gewonnen werden. Es ist schwierig zu sagen, ob solch ein Spiel jemals aus der Beta-Phase austritt.

2.4 Debugger

> „Ein Debugger ist ein Programm das Bugs, also Programmierfehler, aufspürt. Es wird während der Entwicklung und Wartung von Software eingesetzt [...], um Bugs zu finden. Der Programmierer setzt den Debugger während der Entwicklungsarbeiten ein und kann damit den Inhalt seiner Variablen untersuchen. Außerdem ist es möglich das Programm schrittweise auszuführen, um so Fehler im Ablauf aufzuspüren. Damit ein Programm debuggt werden kann, muss es Debug-Informationen enthalten. Das sind spezielle Zusatzinformationen anhand derer der Debugger den Aufbau des Programms ermitteln kann.“ [2]

Ein Debugger besitzt meist die Möglichkeit, Brakepoints zu setzen. Das sind Haltepunkte, welche manuell im Code platziert werden können. Beim Erreichen dieser, pausiert die Applikation und der Entwickler kann sich die Variablen zu diesem Zeitpunkt anschauen. Auch ein zeilenweiser Fortlauf ist möglich. Hierdurch können Abläufe genau nachverfolgt werden.

2.5 Adobe Flash

> „Adobe Flash ist eine Entwicklungsumgebung von Adobe Systems. Der Hauptfokus liegt auf der Erstellung von RIA (Rich Internet Applications[2]), welche durch kombinieren von Grafiken, Animationen, Video und Ton für eine erweiterte Internetnutzererfahrung sorgt.“ [3]

Der Begriff Flash wird in vielerlei Hinsicht für unterschiedliche Sachverhalte genutzt. **Flash Professional** ist zum Beispiel ein Programm von Adobe, vorrangig zum Erstellen von Animationen und Vektorgrafiken. **Flash Builder** ist die integrierte Entwicklungsumgebung für Programmierer. Der **Flash Player** ist ein Browser plug-in um die Laufzeitumgebung für Flash Anwendungen im Internet bereitzustellen. Für die Veröffentlichung für Desktop-Anwendungen ist **AIR** (**A**dobe **I**ntegrated **R**untime) zu verwenden.

Flash war viele Jahre Spitzenreiter was Browseranwendungen und Spiele angeht. Das ist eine Folge der hohen Verbreitung von über 95%. Doch die Erfolgszeit geht dem Ende nahe, derzeit haben nur noch rund 60% [5] der Internetnutzer den Flashplayer installiert. Gründe hierfür sind zum einen HTML5, welches gleiche Features ohne Plugins unter besserer Performance erbringt. Des Weiteren unterstützen die aktuellen Smartphones keine Flashinhalte mehr. Anfänglich galt dies nur für Apple-Nutzer, mittlerweile jedoch auch für Android-Geräte. Dennoch ist es möglich, mit Hilfe von AIR Flashinhalte zum Laufen zu bekommen.

Die meisten Webseiten und Werbebanner, welche mit Flash umgesetzt wurden, sind mittlerweile durch HTML5 ersetzt. Dass HTML5 erst letztes Jahr vom W3C standardisiert wurde, ist wohl auch ein Grund dafür, warum sich der FlashPlayer so lange auf dem Markt halten konnte. Selbst Internetgigant Google ersetzte nun seinen Flash-Videoplayer auf Youtube und hat ihn an die neuere Technik angepasst.

Doch es gibt auch Bereiche in dem Flash noch aktiv und fast konkurrenzlos genutzt wird. Hierzu zählt die Erstellung von Benutzeroberflächen von hochkarätigen Spielen, da hierbei Vektorgrafiken genutzt werden. Diese können für jede beliebige Auflösung in der Größe angepasst werden und somit wirken die Grafiken immer scharf ohne viel Speicher zu verwenden. Was bei der Verwendung von 4K-Bildschirmen schnell mehrere hundert MegaByte sein können.

[2] RIA - Internetanwendung mit vielfältigen Interaktionsmöglichkeiten, z.B. 3D-Effekte, Animationen

2.6 ActionScript 3

Für die Programmierung unter Flash wird das sogenannte ActionScript genutzt. Diese objektorientierte Skriptsprache wurde entwickelt, um RIA dem Flash Player bereitzustellen. ActionScript ähnelt sehr stark JavaScript, was wohl daran liegt, dass beide auf dem selben ECMA-Script Standard aufbauen.

Im Jahre 1999 wurden mit ActionScript 1 Flash-Inhalte interaktiv. Diese Sprache wird ab Flash Player 4 unterstützt und enthält rudimentäre Funktionen, entspricht jedoch schon dem ECMA-Script Standard.

Vier Jahre später kamen mit der Einführung von ActionScript 2.0 und dem Flash Player 6 erste objektorientierte Ansätze hinzu. Dennoch baut es im Inneren weiterhin auf Actionscript 1.0 auf. Deswegen bleiben auch gravierenden Performanceverbesserungen aus.

In der Planungsphase für den Flash Player 9 wurde begonnen, die AVM (ActionScript Virtual Machine) komplett zu überarbeiten. Das Ergebnis war AVM2, eine optimierte virtuelle Maschine, welche auf ActionScript 3 ausgerichtet ist. Dennoch wird auch die ältere AVM1 weiterhin unterstützt um eine Abwärtskompatibilität zu gewährleisten.

2.6.1 Metadata

Annotations (Anmerkungen) sind eine Form der Code-Strukturierung, welche in ActionScript an verschiedenen Stellen verwendet werden. Darüber hinaus dienen sie nicht nur der Ordnung des Quellcodes, sondern ermöglichen es, Werte als Parameter an Klassen und Funktionen anzuhängen.

```
1 [SWF( width="600", height="400", frameRate="60" )]
2 public class Main extends Sprite
3 {
4    ...
```

Listing 2.1: Metadata

Im Codebeispiel wird mithilfe eines Metatags die Größe und die Framerate[3] des Anwendungsfensters festgelegt.

Es ist durchaus möglich eigene Metatags zu definieren. Dies kann von Bedeutung sein, wenn man einigen Klassen oder Methoden Eigenschaften anhängen möchte. Diese

[3] Framerate - Anzahl der Einzelbilder welche in einem gewissen Zeitraum, meist einer Sekunde, angezeigt werden. Erst durch eine entsprechende Framerate entsteht die Illusion einer Bewegung.

Anmerkungen sind auch während der Laufzeit aufrufbar, z.B. über den Befehl *describeType()*. Beim Verwenden von benutzerdefinierten Metatags ist darauf zu achten, dass diese in den Compiler-Options vermerkt sind.

```
1 -keep-as3-metadata+=Metadata
```

Dies muss jedoch nicht bei jeder Verwendung des Frameworks wiederholt eingestellt werden.

2.6.2 Reflection

Reflection bezeichnet den Prozess, programmspezifische Daten zur Laufzeit auszuwerten, um mithilfe dieser das Verhalten zu beeinflussen. Dabei können auch Klassen analysiert werden, um einen Bauplan derer zu erhalten. Mithilfe des Bauplans können Klassen serialisiert werden, um sie anschließend über eine Datenverbindung verschicken zu können.

Serialisierung ist der Vorgang die Struktur einer Klasse zu beschreiben, um sie anschließend in eine serielle Form zu überführen. Das schließt auch eventuelle Substrukturen ein, welche mit dieser Klasse verknüpft sind.

> Der Begriff der Serialisierung wird im Zusammenhang mit objektorientierten Anwendungen und der persistenten Speicherung der Daten verwendet, die mit einem Objekt assoziiert werden. [...] Objekte können in serialisierter Form gespeichert und wieder ausgelesen werden. Anwendung findet die Serialisierung bei der Übertragung von Zuständen von einem Rechner zu einem anderen oder generell in verteilten Software-Systemen. [9]

Serialisierung wird genutzt, um Klassenstrukturen abzuspeichern oder über eine Verbindung mit anderen Applikationen zu versenden. Das Analysieren von Klassenstrukturen kann in ActionScript mithilfe von *describeType()* umgesetzt werden. *describeType()* erzeugt eine XML welche das Objekt beschreibt.

Dieser Vorgang wird immer dann nötig, wenn einer anderen Applikation Klassendefinitionen fehlen. Bei dem zu erstellenden Debugger ist dies der Fall. Dieser wird in verschiedenen Projekten eingesetzt und kennt daher verwendete Klassen nicht.

2.7 Flex

> „Das Entwicklungsframework Adobe Flex dient der Erstellung von interaktiven Benutzeroberflächen für Web- und Desktopanwendungen [...]. Die Ba-

> sis einer Flex-Applikation sind die Markup-Sprache MXML und die objektorientierte Programmiersprache ActionScript 3.0.“ [12]

2008 stellte Adobe das SDK als Open-Source zur Verfügung. Die Rechte wurden an Apache übertragen, welche weiter an Flex entwickeln. Durch Flex können schnell Benutzeroberflächen erstellt werden, welche verschiedene Daten präsentieren.

Ausgangspunkt jeder Flex-Anwendung ist die sogenannte Macromedia Flex Markup Language (MXML). Diese deklarative Markup-Sprache wird vor jedem Compile-Vorgang in Action Script 3.0 Code umgewandelt. Sie dient zur Trennung von Präsentationsschicht und der Businesslogik. Flex-Anwendungen sind plattformunabhängig und benötigen nur den Flash Player 9.x oder höher.

Der nachfolgende Codeblock zeigt eine vereinfachte MXML-Datei. Zu beachten ist: Da Flex-Code intern in AS3 umgewandelt wird, besteht auch jederzeit die Möglichkeit, AS3-Code einzufügen.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <s:WindowedApplication xmlns:fx="http://ns.adobe.com/mxml/2009"
3    xmlns:s="library://ns.adobe.com/flex/spark"
4    xmlns:mx="library://ns.adobe.com/flex/mx">
5
6    <fx:Script>
7      <![CDATA[
8        // ActionScript 3 Code
9      ]]>
10   </fx:Script>
11
12   <fx:Style>
13         @namespace s "library://ns.adobe.com/flex/spark";
14   </fx:Style>
15
16 </s:WindowedApplication>
```

Listing 2.2: MXML Standart-Datei

2.7.1 Oberfläche

Flex verwendet Komponenten um Benutzeroberflächen zu gestalten. In diesem Falle sind die Komponenten grafische Elemente, welche bereits notwendige Funktionen enthalten. So zum Beispiel eine Checkbox, eine Schaltfläche mit Text oder eine Grafik. Mit Flex 4 sind zahlreiche neue Möglichkeiten, dank Spark, hinzugekommen. Diese Komponenten- und Skinning[4]-Architektur ermöglicht CSS[5], verschiedene Zustände (states), Animationen, Text, Grafiken und Layouts zu nutzen.

[4] skinning - Möglichkeit die Darstellung von Anzeigen wie Buttons nach Belieben zu individualisieren
[5] Cascading Style Sheet - beschreibt das Aussehen der Struktur von HTML-Seiten

Spark basiert auf dem zuvor genutzten System MX. Dadurch ist ein Umstieg nicht sehr kompliziert. Es gibt Container, welche jede Art von Komponenten beinhalten kann. Da Komponenten aus beiden Systemen auf der selben Klasse aufbauen, *mx.core.UIComponent*, können diese auch simultan in einer Anwendung genutzt werden. Zur Anordnung der Elemente werden Layouts genutzt. Diese können auch zur Laufzeit angepasst werden. [7]

2.7.2 Data binding

Flex bietet die Möglichkeit, Werte von Objekten direkt an bestimmte Oberflächenelemente zu binden, das sogenannte data binding. Dies hat den Vorteil, das GUI-Elemente bei Wertveränderung direkt aktualisiert werden.

Es gibt drei Möglichkeiten, Daten an Komponenten zu binden. In der ersten werden geschweifte Klammern benutzt, um den zu verknüpfenden Wert anzugeben.

2.8 Ash-Framework

Die nachfolgenden Kapitel beschäftigen sich mit dem Ash-Framework von Richard Lord. Anders als bei reinen objektorientierten Ansätzen, werden hier keine Objekte behandelt, sondern Daten (Komponenten) und Subroutinen (Systeme), welche an diesen Veränderungen vornehmen.

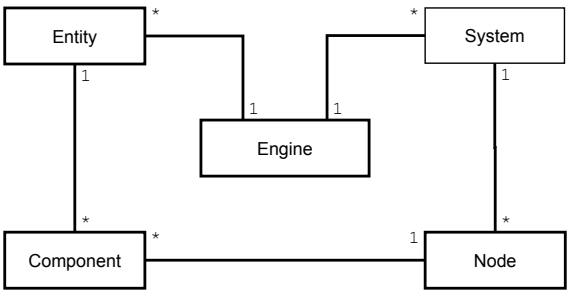

Abbildung 2.5: Ash-Framework Architektur [6]

Grundlegend in dieser Architektur sind die Systeme und die Komponenten. Dabei sind Komponenten reine Wertehalter und bestimmen dadurch einen Zustand. Die Systeme reagieren in Abhängigkeit der Zustände und führen entsprechende Logiken aus. Entities repräsentieren eine Sammlung von Komponenten, welche wiederum ein Objekt beschreiben.

Nodes enthalten auch eine Anzahl an Komponenten. Im Gegensatz zum Entity werden diese genutzt, um wie bei einem Filter auf alle Entities zurückzugreifen, welche die

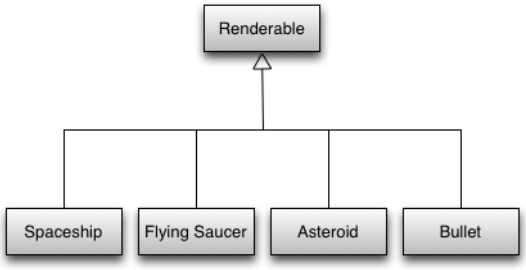

Abbildung 2.6: GameArchitecture Example 1 [6]

Bedingungen erfüllen. Diese werden in Systemen weiterverarbeitet. Da der Node und damit die Komponenten des Objektes bekannt sind, ist kein Cast[6] mehr nötig.

Systeme dienen zur Auskoppelung der Logik von Klassen. Sie arbeiten getrennt voneinander und verändern Daten, ohne zu wissen, was mit diesen vorher oder nachher geschehen ist. In einem System wird nur eine logische Operation ausgeführt. Für weitere Logiken wird empfohlen, weitere Systeme anzulegen.

Einige Beispiele von Systemen in einem Spiel:

- Render-System um Spielelemente auf dem Bildschirm anzuzeigen.
- Physik-System, welches die Position von Spielelementen anhand von physikalischen Gegebenheiten anpasst.
- AI-System, welches nicht spielbaren Einheiten Logik verleiht.
- Eingabe-System, welches die Position von Spielelementen anhand der Eingaben anpasst.

Beim Entwickeln von Spielen kann es nach einiger Zeit schnell vorkommen, dass einige Spielobjekte eine starke Vererbungshierarchie voraussetzen. Projekte werden so schnell unübersichtlich, bestehen nur noch aus Ausnahmefällen und auf Änderungen kann nur langsam reagiert werden.

Für ein Spiel werden in der Regel Objekte gebraucht, welche gezeichnet werden können. So ist eine *Renderable* Klasse nötig um von dieser entsprechenden Elemente zu erben.

Meist ist dennoch mehr nötig als nur statische Objekte. Aus diesem Grund wird eine Klasse *Moveable* erstellt. Da sich nicht alle Objekte bewegen brauchen, wird hierfür nun eine Verzweigung nötig. Doch was wenn das Gamedesign auch bewegliche, unsichtbare Gegenstände vorsieht? Schon nach wenigen neuen Spielelementen wird der Klassenbaum komplex und sieht eventuell wie in Abbildung 2.7 aus.

[6] Cast - Bestimmen bzw. Umwandeln von einem Klassentyp in einen anderen

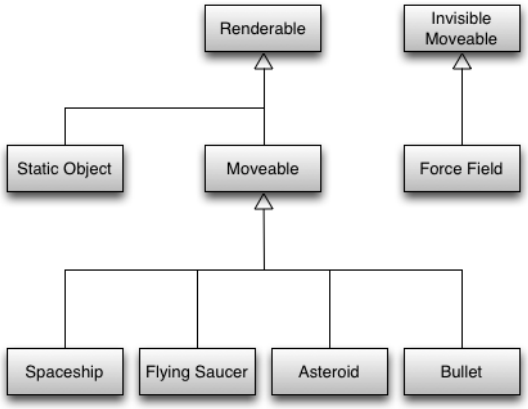

Abbildung 2.7: GameArchitecture Example 2 [6]

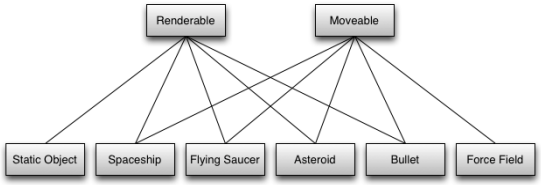

Abbildung 2.8: GameArchitecture Example 3 [6]

Würde das selbe Spiel mit einem Entity-System realisiert, so bräuchte man nur zwei Komponenten. Diese können anschließend nach Belieben auf die Entities angewandt werden. (Abbildung 2.8)

Da der Kernpunkt des Frameworks die Engine ist, kann hier auch auf alle aktiven Entities und Systeme zugegriffen werden.

3 Konzeption

In folgendem Kapitel werden mehrere Ansätze für die Kommunikation zwischen den beiden Applikationen, sowie die Darstellung und Manipulation der Daten gesucht.

3.1 Verbindung

Eine Verbindung zwischen zwei Air / Flashanwendungen ist essentiell. Diese sollte einfach aufgebaut und ohne weitere Einschränkungen oder Einstellungen nutzbar sein.

3.1.1 SharedObject

Ein SharedObject ist ein Objekt, welches persistent lokal auf dem Rechner gespeichert wird. Dadurch können Daten über einen längeren Zeitraum von der selben Applikation genutzt werden oder mehrere Flashinstanzen gemeinsam das Objekt nutzen.

SharedObjects sind vergleichbar mit Cookies im Browser, haben jedoch diverse Vorteile. SharedObjects besitzen technisch gesehen kein Verfallsdatum, sie können größere Datenmengen speichern und es können native ActionScript-Klassen gespeichert werden.

Damit mehrere Applikationen auf das selbe SharedObject zugreifen können, müssen sie auf dem gleichen Client genutzt werden und von der selben Domain aufgerufen werden. Die Daten werden dann in einem Pfad wie etwas
C:/Benutzer/'NutzerName'/AppData/Roaming/'DomainName'/'Applikation'/xxx.sol
gespeichert. Dies funktioniert einwandfrei, wenn die Flashdateien auf einem Server gespeichert sind und über den Browser geöffnet werden. Bei einer Verwendung unter Air über den Desktop ist es aus Sicherheitsgründen nicht möglich, auf das selbe Verzeichnis zuzugreifen.

Zu beachten ist ebenfalls, dass Zugriff von mehreren Seiten auf ein und die selbe Datei besteht. Dadurch muss sichergestellt werden, dass dieser Zugriff nicht zur selben Zeit auftritt. Des Weiteren leidet beim Datenaustausch über eine gespeicherte Datei immer die Geschwindigkeit. [4]

3.1.2 LocalConnection

Zwei Flash Applikationen können auch mithilfe einer LocalConnection direkt miteinander kommunizieren. Einzige Bedingung hierfür ist, dass sie auf dem selben Client laufen.

Dabei spielt es weder Rolle, ob es sich um eine Libary oder Anwendung handelt. Es ist auch nicht relevant, ob eine Version im Browser läuft und die andere nicht.

3.1.3 Sockets

Anders als bei einer LocalConnection müssen die Applikationen bei einer Verbindung über Sockets nicht auf dem selben Client laufen. Dies kann zum Beispiel ein Vorteil sein, wenn das Spiel auf einem mobilen Endgerät und der Debugger auf dem PC läuft.

Bei Socketverbindungen findet die Kommunikation generell über die Netzwerkkarte statt. Ob dies Auswirkungen auf die Übertragungsgeschwindigkeit hat, soll später ermittelt werden. Die Verbindung findet mithilfe des Übertragungsprotokolls TCP statt.

> Das Transmission Control Protocol (TCP) ist ein verbindungsorientiertes Transportprotokoll für den Einsatz in paketvermittelten Netzen. Das Protokoll baut auf dem IP-Protokoll auf, unterstützt die Funktionen der Transportschicht und stellt vor der Datenübertragung eine gesicherte Verbindung zwischen den Instanzen her. [11]

Da Daten segmentiert werden, um diese als Datenpakete zu versenden, kann es passieren, dass einzelne Nachrichten nicht im Ganzen direkt übertragen werden. Auch das Gegenteilige kann passieren und mehrere Nachrichten werden zu einem Paket gebündelt. Die Größe eines solchen Pakets beträgt maximal 65 kB.

Es ist auch möglich, eine Socket-Verbindung mithilfe von UDP aufzubauen.

> Das User-Datagram-Protokoll (UDP) ist ein verbindungsloses Transportprotokoll für den Datenaustausch zwischen Rechnern. Das UDP-Protokoll wurde definiert damit Anwendungsprozesse direkt Datagramme versenden können und damit die Anforderungen an transaktionsorientierten Verkehrs erfüllen. Das UDP-Protokoll baut direkt auf dem darunter liegenden IP-Protokoll auf und zeichnet sich durch seinen geringen Overhead und die kurzen Latenzzeiten aus.

Dieses Protokoll sollte Verbindungen mit geringerer Latenz[7] aufbauen. Datenpakete können jedoch in unterschiedlicher Reihenfolge ankommen. Pakete, welche bei der Verbindung verloren gingen, werden nicht erkannt. Datenpakete werden nicht automatisch aufgeteilt. Werden Pakete verschickt, welche die maximal Größe überschreiten, so werden sie ohne Rückmeldung einfach abgeschnitten.

[7] Latenz - Verzögerte Reaktion auf einen Reiz oder ein Signal

3.2 Datenpaket

Die Daten müssen natürlich irgendwie zwischen den beiden Flash-Instanzen übermittelt werden. Dazu sollten die gesendeten Informationen der Objekte so gering wie möglich ausfallen. Um die angezeigten Daten im Debugger aktuell zu halten, werden die Daten in kurzen Abständen hin und her geschickt. Der Serialisierungsvorgang darf demnach auch nicht allzu komplex ausfallen, um den Spielfluss nicht zu beeinflussen.

Bevor die Daten versandt werden können, müssen sie zu einem Datenpaket geschnürt werden. Beim Versenden kann es vorkommen, dass Nachrichten geteilt oder auch mit nachfolgenden verbunden werden. Am besten hierfür eignet sich ein *ByteArray*. Es verfügt über die Möglichkeit, Standard-Typen wie *Int*, *String* und sogar *Object*'s direkt hinzuzufügen. *ByteArray*'s können zusätzlich noch komprimiert werden, um die Größe weiter zu verringern.

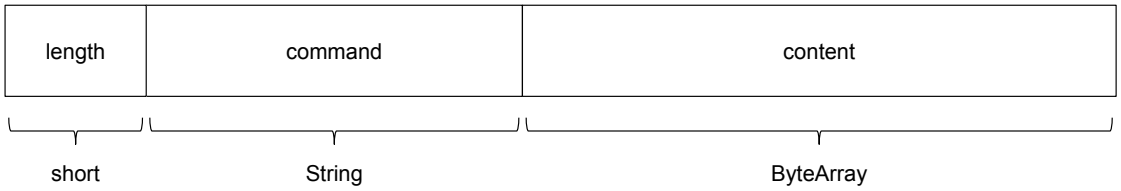

Abbildung 3.1: Package-Struktur

In Abbildung 4.3 soll ein möglicher Aufbau eines Datenpaketes gezeigt werden. Zu Beginn steht die Länge der nachfolgenden Nachricht in Bytes. Es kann vorkommen, dass versandte Nachrichten geteilt wurden und deshalb noch nicht die ganze Nachricht übertragen worden ist. In diesem Fall vergleicht der Empfänger die Länge und wartet gegebenenfalls auf weitere Daten. Ebenfalls möglich ist, dass mehrere Nachrichten beim Senden miteinander verbunden und an einem Stück übermittelt werden. Auch hier wird dank der Größeninformation das Nachrichtenende ermittelt.

Nachfolgend bietet ein Kommando in Form eines *String* die Möglichkeit, Nachrichten zu differenzieren. Je nach Art können diese anschließend unterschiedlich behandelt werden. Hierdurch kann nachfolgender Inhalt von verschiedener Länge und Typkombination sein. Denn nicht immer müssen komplette Objekte übergeben werden. So reicht es, beim Pausieren der Applikation nur entsprechende Kommandos zu senden. Hierfür sind keine weiteren Werte notwendig. Anhand des Kommandos werden die Nachrichten beim Empfänger unterschiedlich interpretiert.

3.3 Monitoring

Monitoring der Werte kann in Flex mithilfe von Data-Binding gelöst werden. Hier bei wird zwischen einer einfachen und einer Zwei-Wege-Bindung unterschieden. Ersteres

spiegelt einen Wert nur in eine Richtung. Er selbst wird nicht von dritten geändert. Eine solche Bindung wird häufig genutzt um Werte über ein Textfeld oder eine ähnliche Benutzeroberfläche für den Nutzer sichtbar zu machen. Dies kann zum Beispiel die Uhrzeit sein. Bei jeder Änderung, in diesem Falle jede Minute, wird das Textfeld darüber informiert das ein neuer Wert vorliegt. Logischer Weiße kann aber der Nutzer nicht die Uhrzeit verändern.

Anders ist es allerdings bei einem Warenkorb auf einer Versandhändlerseite. Hier werden oftmals alle Waren im Einkaufkorb in einer Liste angezeigt. Hier kann der Nutzer sowohl die Anzahl der Produkte ändern, als auch einzelne Produkte löschen. Er ändert aktiv die gespeicherten Daten im Hintergrund. Dafür ist eine Zwei-Wege-Bindung nötig. [**?**]

3.4 Manipulation an der Engine

Welche Möglichkeiten für Veränderungen am Ash-Framework möglich sind, soll in diesem Kapitel beleuchtet werden. Wie in Kapitel 2.8 erläutert besteht das Ash-Framework vorrangig aus folgenden Klassen, *Engine*, *System*, *Entity*, *Component* und *Node*.

3.4.1 Engine

Die Engine soll als Einstiegspunkt dienen. Deshalb wird sie wohl auch ein Parameter für den Debugger werden. Die Engine enthält Informationen über diverse aktive Elemente wie etwa alle Entities und Systeme. Durch Hinzufügen oder Entfernen ist es möglich, das Verhalten zu ändern. Das Entfernen des Hittest-Systems führt zum Beispiel dazu, dass keine Kollision zwischen den Objekten mehr stattfindet.

Grund hierfür ist, dass die Engine alles steuert und als einziges weiß, welche Systeme und Entities allumfassend existieren. Im Grunde koordiniert die Engine alle Aufgaben und Zugehörigkeiten.

Die Applikation sollte so auf das Ash-Framework aufbauen, dass bei Entfernen aller Systeme das Spiel in einer Art Pausemodus verharrt.

Entfernte Objekte müssen zwischengespeichert werden, wenn diese später erneut reaktiviert werden sollen.

3.4.2 Entity

Auf viele Eigenschaften eines Entity kann nach erfolgreichem Extrahieren aus der Engine zugegriffen werden. Dies stellt hauptsächlich Components dar, welche auf ähnliche

Art und Weise entfernt und hinzugefügt werden können.

Die Anzahl der Entities kann von Spiel zu Spiel stark schwanken. Eine Menge von mehreren tausend kann schnell erreicht werden. Eine häufige Änderung an den hinzugefügten Components kann auch erfolgen.

3.4.3 Component

Komponenten zu beeinflussen ist hingegen etwas schwieriger. Das Auslesen der verwendeten Komponenten kann über die Entities erfolgen. Nach einem Durchlauf und Abgleich erhält man eine Liste. Doch was ist mit Komponenten, welche noch nicht oder nie genutzt werden?

Es ist nicht möglich, alle Components anhand des Types zu bestimmen, da keine reale Component-Klasse existiert. Das kann durch Erzeugen einer eigens dafür angelegten Klasse oder Interface vermieden werden. Das Problem hierbei ist, dass schon vorhandene Projekte angepasst werden müssen, bevor sie das Framework in vollem Umfang nutzen können.

Mithilfe von Metatags könnte eine gewisse Differenzierung erfolgen. Wie bereits weiter oben beschrieben ist dies ohne große Umstände möglich. Ähnlich wie bei dem ersten Ansatz müssten bereits erstellte Projekte angepasst werden.

Eine weitere Möglichkeit ist es, die Klassennamen zu analysieren. Dies würde das Bearbeiten von Projekten verhindern, wenn in diese eine einheitliche Namenskonvention angewandt wurde, zum Beispiel ”RenderComponent„, ”CollisionComponent„, ... Hier besteht aber immer die Gefahr, dass nicht gewollte Objekte durch den Suchalgorithmus fallen, nur weil sie zufällig Component im Namen enthalten.

Eine Art Registriersystem könnte dafür sorgen, dass nur ein minimaler Eingriff in vorhandene Projekte nötig ist. Lediglich müssen die zu verwendeten Components beim Debugger registriert werden.

3.4.4 Node

Da Nodes nur eine Bündelung von Components darstellen, wirkt es, als ob sie keine große Relevanz für den Debugger besitzen. Dennoch könnte diese Klasse sehr nützlich sein um Ergebnisse zu filtern und die Benutzeroberfläche intuitiver zu gestalten. Eine direkte Veränderung kann nicht vorgenommen werden, da dies keine Auswirkungen auf das Spiel haben würde.

3.4.5 System

An Systemen kann von Außen auch nicht viel verändert werden. Lediglich das Aktivieren und Deaktivieren wird über den Debugger gesteuert.

3.5 Reflection

Reflection wird zur Serialisierung der Klassen benötigt. Diese müssen analysiert werden und in ihre Einzelteile zerlegt. Notwendig ist dies jedoch nur bei den Entities und den Components.

Die Analyse findet mit Hilfe des *describeType()*-Befehls statt. Dieser durchläuft eine Klasse und speichert Informationen dieser in einer XML-Struktur. Dabei können sowohl Variablen als auch Methoden und sogar Metadaten ausgewertet werden. Zu beachten ist hierbei jedoch, dass keine mit *private* oder *protected* gekennzeichneten Werte analysiert werden können. Eine Verwendung von *getter* und *setter* ist jedoch möglich.

Hier ein Beispiel einer Analysierten Klasse mit *describeType()*.

```
1 public class Person{
2   public var name:String;
3   public var age:int;
4   public var alive:Boolean;
5 }
```

Listing 3.1: Beispiel-Klasse

```
<type name="Person" base="Class" isDynamic="true" isFinal="true"
   isStatic="true">
2 <extendsClass type="Class"/>
3 <extendsClass type="Object"/>
4 <accessor name="prototype" access="readonly" type="*" declaredBy=
   "Class"/>
5 <factory type="Person">
6   <extendsClass type="Object"/>
7   <variable name="alive" type="Boolean">
8     <metadata name="__go_to_definition_help">
9       <arg key="pos" value="95"/>
10    </metadata>
11  </variable>
12  <variable name="name" type="String">
13    <metadata name="__go_to_definition_help">
14      <arg key="pos" value="47"/>
15    </metadata>
16  </variable>
17  <variable name="age" type="int">
18    <metadata name="__go_to_definition_help">
19      <arg key="pos" value="73"/>
```

```
20        </metadata>
21      </variable>
22      <metadata name="__go_to_definition_help">
23        <arg key="pos" value="24"/>
24      </metadata>
25    </factory>
26  </type>
```

Listing 3.2: describeType-Anwendungsbeispiel

Der Debugger kennt die verwendeten Klassen nicht. Deswegen könne Befehle wie *registerClassAlias()* nicht verwendet werden. So besteht die Notwendigkeit, die Klassen bis auf den kleinsten bekannten Datentyp herunter zu brechen. Sprich, aus einem *Vector3D* wird ein Objekt mit 5 Unterobjekten vom Typ Number.

3.6 Benutzeroberfläche

Die ausgewerteten Daten müssen in einer Benutzeroberfläche dargestellt werden. Mit Flex ist es möglich, grafische Komponenten modular zu gestalten. So können die einzelnen Bereiche, auch Fenster genannt, unabhängig voneinander arbeiten. Folgend wird auf die wichtigsten Bedienoberflächen eingegangen und deren Anforderungen ermittelt.

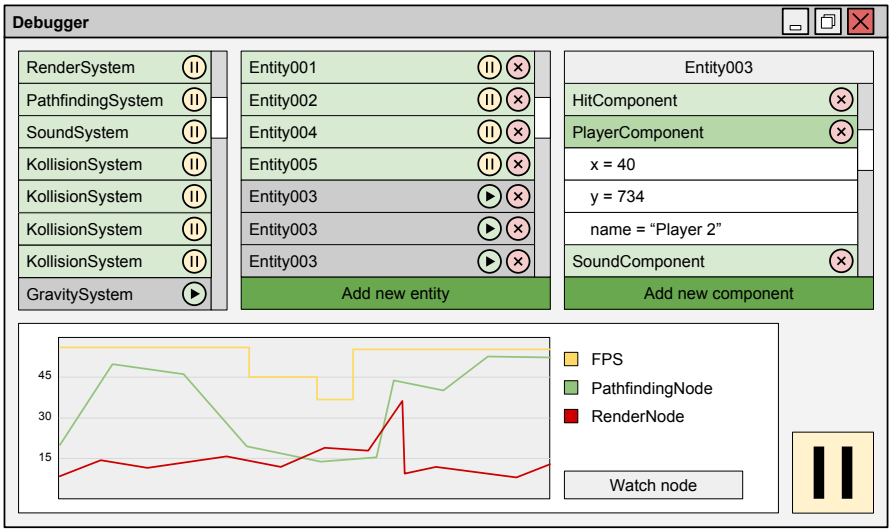

Abbildung 3.2: Mockup Debugger

3.6.1 Entity-Fenster

Das Entity-Fenster zeigt alle aktiven und inaktiven Entities im Spiel. Diese werden in einer vertikalen Liste angezeigt. Da davon ausgegangen werden kann, dass es sich

hierbei um eine beträchtliche Anzahl handeln kann, bietet eine Liste die Möglichkeit, alle Daten zu repräsentieren ohne das Layout zu sprengen.

Abbildung 3.3: Entity-Fenster

In diesem Fenster muss es dem Nutzer möglich sein, Entities zu beobachten, sie zu de- und aktivieren, als auch sie zu löschen oder neue zu erstellen. Außerdem soll bei einem Klick auf das entsprechende Entity dessen Komponenten in einem separaten Fenster angezeigt werden. Die Entities werden dem Alphabet aufsteigend sortiert. Da der Name automatisch vom Ash-Framework vergeben wird, hat dieser immer folgende Form: Entity + id. So zum Beispiel “Entity56„. Daraus folgt, dass eine Sortierung nach Namen automatisch einer Sortierung nach Erstelldatum entspricht. Wird ein Eintrag deaktiviert, so erscheint er in gräulicher Form am Ende der Liste um die Übersichtlichkeit zu bewahren.

Das De- und Aktivieren ist über die entsprechenden Schaltflächen am Rande des Feldes möglich. Deaktivierte Entity-Objekte werden zwischengespeichert und aus der laufenden Engine entfernt. Sie sind für das Spiel nicht mehr existent, können später jedoch reaktiviert werden. Wird ein Entity nicht mehr benötigt, kann es mit der X-Schaltfläche endgültig gelöscht werden.

Es kann durchaus vorkommen, dass ein weiteres Entity hinzugefügt werden soll. Dies kann über die untere Schaltfläche getätigt werden. Hiernach wird ein neues Entity erzeugt und der Engine hinzugefügt. Nun können Komponenten hinzugefügt werden, um diesem Logik zu verleihen.

3.6.2 Component-Fenster

Im Component-Fenster werden alle Komponenten das selektierten Entity angezeigt. Dieser Bereich stellt ebenfalls in einer Art Baumstruktur alle öffentlichen Variablen der entsprechenden Komponenten dar. Die Werte der Variablen können hier direkt eingesehen und verändert werden. Lediglich die Werte der zuletzt gewählten Komponente sind sichtbar. Wie bei den Entities, können mit der X-Schaltfläche Einträge gelöscht werden.

Über den “**Add new component**„-Button können dem Entity neue Komponenten hinzugefügt werden. Nach dem Klick öffnet sich ein neues Fenster, welche je nach Component gewisse Parameter enthält.

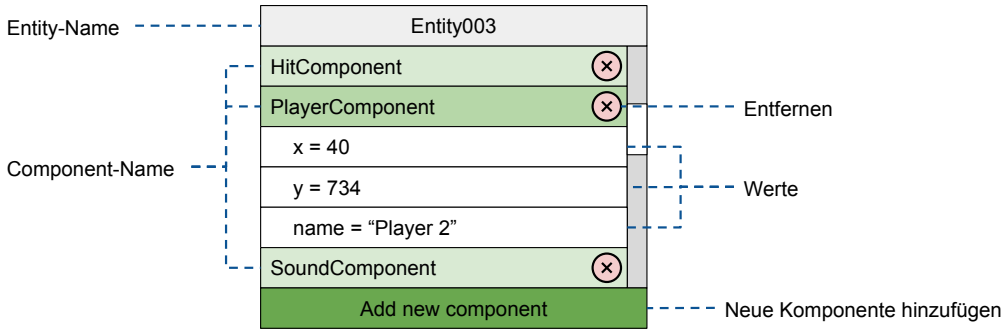

Abbildung 3.4: Component-Fenster

3.6.3 System-Fenster

Das System-Fenster ist recht simpel. Hier werden alle Systeme in einer Liste angezeigt. Diese sind ebenfalls nach dem Alphabet sortiert. Dieses Fenster dient lediglich zur Kontrolle. Laufende Systeme können pausiert und über die selbe Schaltfläche wieder gestartet werden.

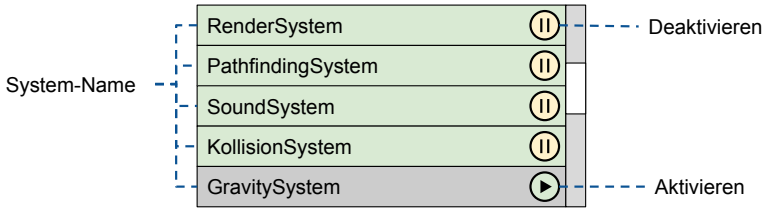

Abbildung 3.5: System-Fenster

3.6.4 Pause-Fenster

Mithilfe des Pausebuttons kann die Applikation gestoppt werden, um in Ruhe Einstellungen vorzunehmen. Es soll möglich sein, zwischen zwei verschiedenen Pause-Modi zu wählen. Ein Modus, bei dem die komplette Flash-Instanz im Ruhezustand verharrt und ein Modus, bei dem alle Systeme temporär aus der Engine entfernt werden.

3.6.5 Performance-Fenster

Das Performance-Fenster liefert in festen Intervallen Information über die aktuelle Spielsession. In einem Liniendiagramm werden zum einen die FPS angezeigt. Zusätzlich

können gewisse Nodes ausgewählt werden, um deren Verwendung darzustellen. Jede neu anzuzeigende Linie wird mit einer anderen Farbe dargestellt. Mit einem Klick auf die aktiven Nodes können diese in der Anzeige ausgeblendet werden. Das Aktualisierungsintervall soll im Debugger bestimmt werden können.

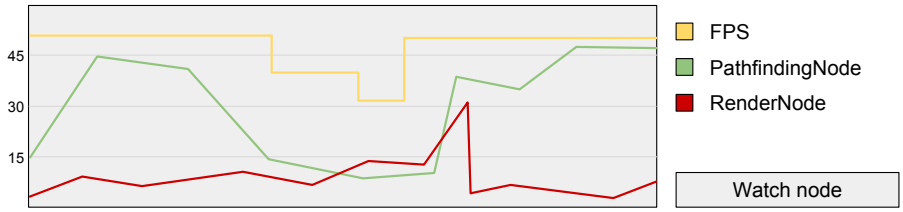

Abbildung 3.6: Performance-Fenster

4 Entwurf

Der Entwurf dient zur genaueren Beleuchtung vorher besprochener Thematiken und Probleme. Hier werden bereits Analysen durchgeführt und Diagramme sowie Strukturen erarbeitet.

4.1 Verbindung

Eine Kommunikation über *SharedObject* ist für solch eine Verbindung aus vielerlei Gründen nicht sinnvoll. Eine Verständigung mithilfe einer Datei, in welche parallel geschrieben wird, führt schnell zu Fehlern und besitzt keine große Geschwindigkeit. Des Weiteren traten beim Test diverse Probleme auf, was die Sichtbarkeit des *SharedObject* angeht. Flash-Applikationen besitzen aus Sicherheitsgründen nur einen Zugriff auf einen bestimmten Ordner und deren Unterordner. Meist haben nur Applikationen von der selben Domain Zugriff auf das gleiche Verzeichnis.

localConnection und *socket*'s bieten ähnliche Features. Das Implementieren beider Varianten ist relativ simpel. Um die Übertragungsgeschwindigkeit zu überprüfen, wurde eine kleine Test-Applikation geschrieben, welche bis zu 10000 Nachrichten hin und her sendet und dabei die durchschnittliche Übertragungsdauer ermittelt. Vor dem Absenden der Nachricht wird mithilfe von *getTimer()* ein Timestamp erzeugt. Sobald die zweite Applikation etwas erhält, schickt sie die selbe wiederum zum Empfänger zurück. Hier wird anschließend abermals ein Timestamp erzeugt und mit dem alten verrechnet. Das Ergebnis wird am Ende noch durch zwei geteilt, da die Zeit für den Hin- und Rückweg gilt.

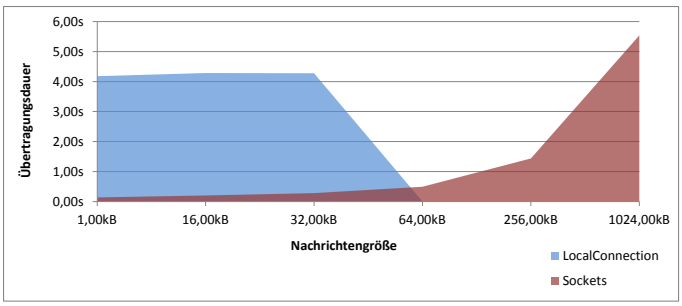

Abbildung 4.1: Verbindungsgeschwindigkeit

Das Ergebnis dieses Tests ist in Abbildung 4.1 einzusehen. Überraschenderweise zeigt sich ein deutlicher Geschwindigkeitsunterschied zwischen den beiden Übertragungsmöglichkeiten. Bei *localConnection* ist ein Anstieg der Übertragungsdauer in Abhängigkeit von der Nachrichtengröße zu sehen.

Wie bereits erwähnt, besitzt eine Verbindung über *Socket*'s die Möglichkeit, Computer übergreifend zu kommunizieren. Leider ist dies nur möglich, wenn sich beide Geräte im selben Netzwerk befinden.

ByteArray's können mit Hilfe von *.compress()* und *.uncompress()* komprimiert werden. Der Komprimierungsvorgang verkleinert die Dateigröße ohne Datenverlust unter Einbusen der FPS. Bei diesem Vorgang wird einer von drei Algorithmen verwendet. Standardmäßig ist das zlib, jedoch kann auch zu deflate oder lzma gewechselt werden.

In Abbildung 4.2 ist jeweils der Übertragungsvorgang mit und ohne komprimierten Daten sowie die Dauer inklusive Komprimiervorgang dargestellt. Hier wird deutlich, dass sich das Komprimieren von *ByteArray*'s bei einer großen Datenmenge lohnt, welche selten versendet wird. Der Mehraufwand der Komprimierung lohnt sich ebenfalls bei Daten, welche sich nie oder kaum ändern. Da ein Debugger möglichst aktuelle Werte anzeigen soll, ist eine hohe Aktualisierungsrate nötig. Auch das große Aufkommen der zu sendenden Daten spricht gegen eine Verwendung von komprimierten *ByteArray*'s.

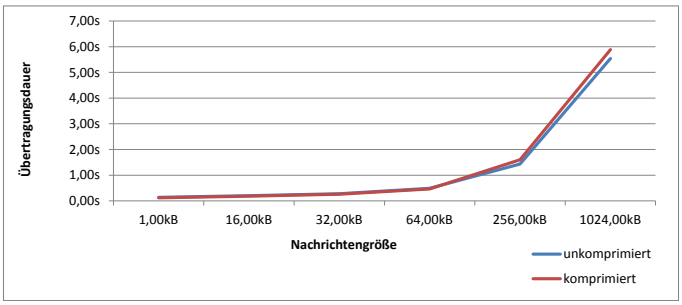

Abbildung 4.2: Übertragungsdauer mit und ohne Komprimierung

4.2 Datenpaket

Die zu versendeten Daten sind von unbekannter Größe. Diese richtet sich je nach Klasse und Variablen, welche diese beschreiben. Deswegen muss das Datenpaket auch anpassbar sein.

4.2.1 Aufbau

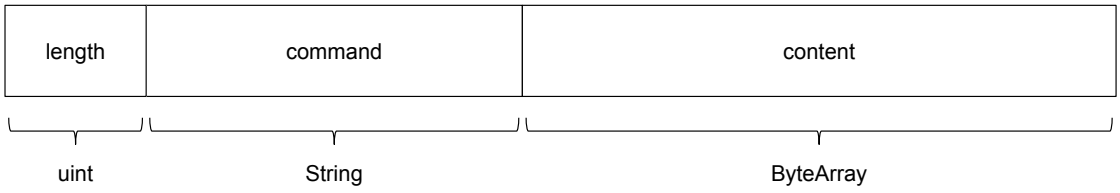

Abbildung 4.3: Package-Struktur

Der Kopf des Paketes gibt an, wie groß der nachfolgende Teil ist. Diese Angabe erfolgt in Byte. Bei Verwendung eines *UnsignedShort* wäre die Maximalgröße 16 bit. Daraus folgt, dass die Paketlänge nicht größer als 64 KByte lang sein darf. Stattdessen würde eine Umstellung auf einen *UnsignedInt* (32 bit) eine Länge von bis zu 4 GByte ermöglichen. So wird sichergestellt, dass ein Paket eine Überlänge erreichen kann.

Gefolgt von dem Kopf ist ein Kommando in Form eines *String*. Dieser wird immer benötigt, um das Paket und daraus folgende Verarbeitung zu beschreiben. In einigen Fällen reicht lediglich ein Kommando ohne nachfolgende Informationen.

Zum Schluss kommt der eigentliche Paketinhalt. Dieser kann verschiedene Typen mit unterschiedlichen Längen enthalten.

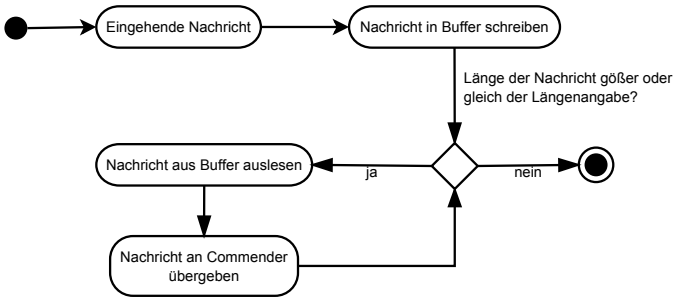

Abbildung 4.4: Aktivitätsdiagramm Nachrichtenempfang

Da ungewiss ist, ob die Nachrichten geteilt oder gebündelt wurden, muss dafür eine Klasse erschaffen werden welche diese Ausnahmen abhandelt. In Abbildung 4.4 wird der Ablauf beim Erhalten einer Nachricht geschildert. Die Klasse liest das eintreffende *ByteArray* ein und speichert es im Buffer[8]. Ist der Buffer anschließend mit genügend Bytes gefüllt, so wird der entsprechende Teil weitergegeben und aus dem Buffer gelöscht. Da es möglich ist, dass mehrere Nachrichten in den Buffer gespeichert wurden, muss dies abermals geprüft werden.

Da bei einem fertigen Spiel einige Tausend Entities in der Engine existieren können, wäre es unklug, diese alle einzeln zu versenden. Das hätte einerseits negativen Einfluss auf die Performance und anderseits geht schnell der Überblick verloren wann welche Daten versendet werden. Der Empfänger muss darauf eingestellt werden einzelne, aber auch gebündelte Nachrichten zu verarbeiten. Dank der einheitlichen Struktur der Nachrichten und dem vorangestellten Längenparameter können Datenpakete wie einzelne Nachrichten abgearbeitet werden.

[8] Buffer - Datenzwischenspeicher, können Daten nicht sofort weiterverarbeitet werden, dann werden sie bis zu ihrem Einsatz temporär gespeichert

4.2.2 Typen

Der zu versendende Inhalt muss nicht immer groß sein oder den gleichen Parametertyp beinhalten. Für die Grundfunktionen reichen folgende Typen, welche mit einer eindeutigen Konstanten als zweiten Parameter differenziert werden können.

- **Command-Send-String** - Der einfachste aller Befehle. Hier wird lediglich eine Zeichenkette übertragen. Dieses Kommando wird nur im Entwicklungsprozess genutzt, da man hiermit schnell Daten zur Überprüfung in der Konsole ausgeben kann.
- **Command-Update-Entity** - Eines der am meist genutzten Befehle, dient zur Übertragung der Entities. Verschickt wird ein JSON-String, welcher den Namen des Entity und deren Zustand (aktiv oder pausiert) enthält.
- **Command-Update-System** - Ist das Äquivalent zum vorherigen Kommando, nur für Systeme.
- **Command-Start-Update** - Dem Debugger werden nur existierende Objekte zugesandt. Er weiß also nicht, wann ein Objekt im Spiel gelöscht wurde. Dies zu erfassen, würde ein kompliziertes Verfahren benötigen. Doch durch diesen Befehl weiß der Debugger, wann ein Updatevorgang startet. Jetzt schaut er nach der Beendigung durch **Command-Finished-Update** selbst, welche Systeme oder Entites nicht aktualisiert wurden und entfernt diese. Der Befehl an sich verschickt keinen zusätzlichen Parameter.
- **Command-Detailed-Info** - Wurde auf Debuggerseite ein Entity ausgewählt, so bittet er das Spiel, ihm zusätzliche Information darüber zu senden, um diese dann anzeigen zu können. Hierbei wird lediglich der eindeutige Entityname mitgesendet.
- **Command-Pause-Application** - Auch dieser Befehl ist Parameterlos und weist das Spiel an, in den Pausezustand zu wechseln. Mit **Command-Resume-Application** wird dieses wiederum zum Leben erweckt.
- **Command-Stats** - Wird genutzt, um Statistiken, wie etwa die aktuellen FPS, zu übermitteln. Hierbei werden die Werte in ein einfaches Object geschrieben und im JSON-Format als Parameter angehängt.

4.3 Monitoring

Den einzelnen Flex-Componenten muss mittels DataProvider ein Verweis auf Inhalt zugewiesen werden. Diese Inhalte befindet sich gebündelt in einer ArrayCollection, welche eine erweiterte Art des Array darstellt, alle in einer Klasse, welche diese Inhalte verwaltet. Diese Klasse kümmert sich darum, das von beiden Seiten die Werte geändert werden können und dies an das Spiel oder die Bedienoberfläche des Debuggers weitergegeben wird.

4.4 Manipulation an der Engine

Der Debugger wird ständig auf die Engine des AshFrameworks zugreifen. Um dies zu zentralisieren und zu optimieren, wird ein Wrapper erstellt. *AMEngine* enthält alle nötigen Funktionen, um auf die Systeme und Entities zugreifen zu können. Des Weiteren speichert er alle pausierten Klassen zwischen und sorgt dafür, dass sie problemlos wieder gestartet werden können.

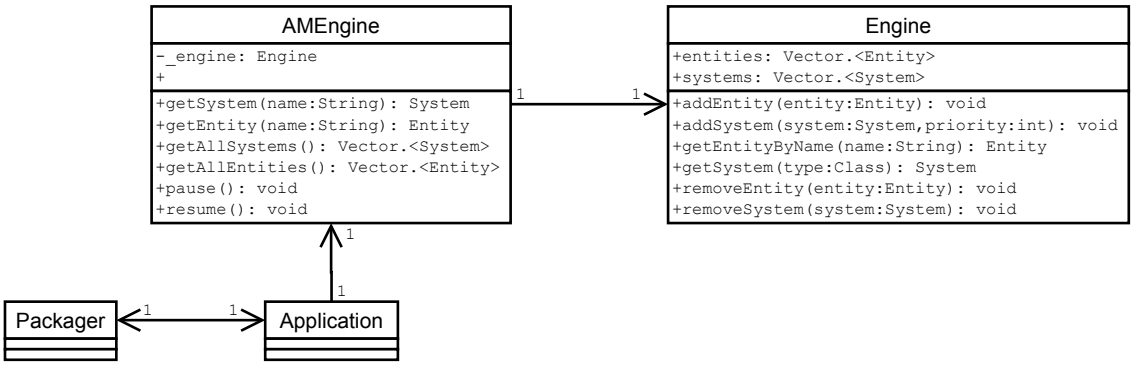

Abbildung 4.5: Klassendiagramm - Beziehung zwischen Applikation und Engine

```
<s:TextInput id="text1"/>
<s:TextInput id="text2" text={text1.text} />
```

Listing 4.1: Code-Beispiel Binding 1

Im zweiten Ansatz wird fx:Binding benutzt.

```
<fx:Binding source="text1.text" destination="text2.text"/>
<s:TextInput id="text1"/>
<s:TextInput id="text2"/>
```

Listing 4.2: Code-Beispiel Binding 2

Auch möglich ist das Verwenden von BindingUtils.bindProperty().

```
<fx:Script>
<![CDATA[
import mx.binding.utils.BindingUtils;
import mx.events.FlexEvent;

protected function initHandler(event:FlexEvent):void
{
BindingUtils.bindProperty(text2,"text",text1,"text");
}
]]>
</fx:Script>
<s:TextInput id="text1"/>
<s:TextInput id="text2" preinitialize="initHandler(event)"/>
```

Listing 4.3: Code-Beispiel Binding 3

4.5 Reflection

Es steht nicht nur *DescribeType* zur Verfügung, um Klassen zu analysieren. Adobe stellt mit *DescribeTypeJSON* eine weitere Möglichkeit. Als Resultat entsteht hierbei keine XML sondern ein JSON-Objekt. Der Große Vorteil hiervon ist die Geschwindigkeit, welche bis zu 5x schneller ist. Und das Resultat kann ähnlich verwendet werden wie die XML.

Dennoch ist dieser Vorgang sehr zeitaufwändig und es empfiehlt sich, die Resultate zu speichern. Das ist jedoch kein Problem, da sich Klassenstrukturen zur Laufzeit nicht ändern. Angebracht hierfür wäre ein Dictionary. Darin können Objekte gespeichert werden und über einen Index aufgerufen werden. Da dieser Index eindeutig sein muss, könnte hierfür der Klassenname mit Pfad verwendet werden.

Diese Baupläne werden benötigt, um Klassen zu serialisieren bevor sie versendet werden können. Hierbei sollen alle Variablen ausgelesen werden, welche Lese- oder Schreibrecht besitzen. Dafür gibt es eine Library namens Spicefactory. Diese nutzt ebenfalls *DecribeTypeJSON* und speichert bereits reflektierte Klassen. Mit Funktionen wie *getMethods()*, *getProperties()* und *getSuperClass()* ist ein einfacher Zugriff auf alle interessanten Werte möglich.

```
1 var ci:ClassInfo = ClassInfo.forClass(Point);
2 var p:Property = ci.getProperty("x");
3 trace("type:     " + p.type.name);
4 trace("readable: " + p.readable);
5 trace("writable: " + p.writable);
```

Listing 4.4: ClassInfo

Ausgabe:

```
1 DEBUG: Add flash.geom::Point to cache for [Domain 1]
2 type:     Number
3 readable: true
4 writable: true
```

Darüber hinaus können auf ähnlichem Wege auch Methoden aufgerufen werden. Diese können sogar Parameter enthalten. Bei erneuter Analyse der selben Klasse *Point* wird statt dem rechenintensiven Aufruf von *DescribeTypeJSON* einfach der gespeicherte Wert aus dem Cache gelesen.

4.6 Benutzeroberfläche

Der große Vorteil von Flex ist wohl die große Auswahl an bereits definierten Benutzeroberflächen. Diese Komponenten können beliebig miteinander kombiniert werden. Daraus entstehen nicht nur einzigartige Designs, sondern auch dynamische Fenster. Das bedeutet es spielt kaum eine Rolle wie groß letztendlich die Applikation ist, die Oberfläche wird sich immer anpassen. Dies kann in Zeiten von mobilen Anwendungen oft eine entscheidende Rolle spielen.

4.6.1 System- und Entityfenster

Von der Funktionalität, sowie Gestaltung unterscheiden sich diese zwei Fenster kaum. Des wegen kann hierfür auch die selbe Component genutzt werden. Eine einfach Liste genügt für diesen Zweck. Eine Liste zeigt, meist vertikal, alle Einträge an. Der Nutzer kann diese markieren und anklicken. Mit einem eigenen Renderer können zusätzlich neben dem Namen des Eintrages, auch noch Grafiken angezeigt werden. Außerdem kann eine Hintergrundfarbe in Abhängigkeit von Eigenschaften eingestellt werden. Überschreitet der Inhalt die Fensterdimensionen, so werden automatisch Scrollleisten erzeugt.

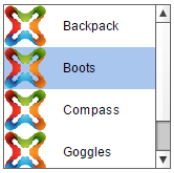

Abbildung 4.6: List

4.6.2 Componentfenster

Dieses Fenster wird erst aktiv, sobald ein Entity ausgewählt wurde. Zu dessen Components werden dann nähere Informationen in einem Tree, zu deutsch Baum, angezeigt. Dieser Name rührt daher, weil alle Elemente Verzweigungen enthalten können, um ihre Kindelemente anzuzeigen. Schließt man ein Element, so werden auch alle untergeordneten Elemente geschlossen. Das sorgt für eine gute Übersichtlichkeit. Auch in diesem Fenster gibt es die Möglichkeit einen eigenen Renderer zu kreieren, welcher das Aussehen und Verhalten ändert.

4.6.3 Pause-Fenster

Das Pausefenster, sowie alle anderen benötigten Schaltflächen, werden durch einen Button realisiert. Eines der simpelsten Interaktionsmöglichkeiten für den User. Hier kann

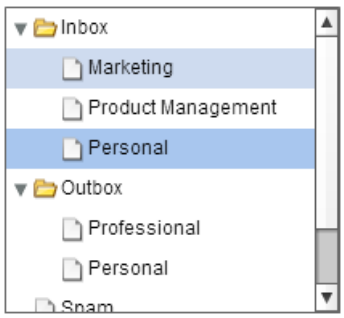

Abbildung 4.7: Tree

sowohl Text als auch eine Grafik angezeigt werden.

4.6.4 Performance-Fenster

Einen Entwickler interessieren auch Information außerhalb des Quelltextes. Etwa wie flüssig das Spiel läuft oder wie viel Speicher es benötigt. Doch ist nicht nur der aktuelle Wert von Bedeutung, sondern auch die vorherigen. Erst mit einer Betrachtung über einen längeren Zeitraum, kann ein eventuelles Problem analysiert werden. Flex bietet hierfür eine Art Liniendiagramm, ein sogenanntes Linechart. In diesem Diagramm können beliebig viele Werte präsentiert werden.

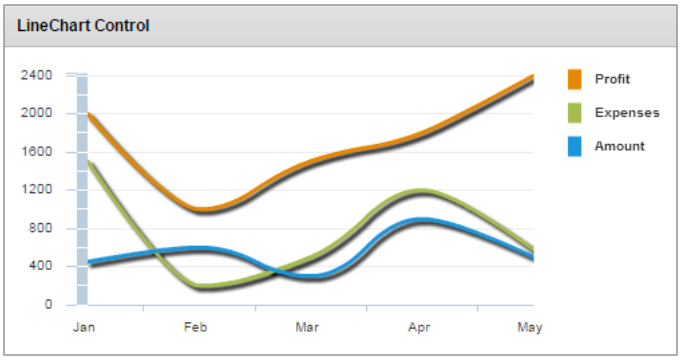

Abbildung 4.8: LineChart

5 Implementierung

In diesem Kapitel werden behandelte Themen und Problemlösungen implementiert. Es folgt eine Erläuterung der Projektstruktur und verwendete Klasse.

Das Programm besteht aus zwei Teilen. Zum einen die eigenständige Bedienoberfläche (AshMonitorApplication), der Debugger. Und zum anderen das ins Spiel zu implementierende FrameWork (AshMonitor). Der Debugger nutzt jedoch auch das Framework, da in vielen Teilen der selbe Code Verwendung findet und gewisse Konstanten auf beiden Seiten einheitlich benannt werden müssen.

5.1 Verbindung

Um eine Verbindung zwischen den Beiden Applikationen zu erstellen, wird eine Socketverbindung benötigt. Der Debugger dient hierbei als Server auf den sich dann theoretisch mehrere Clients verbinden können. In diesem Falle ist es jedoch ausreichend, wenn sich nur ein Client verbinden kann. Um einen Server zu starten, sind lediglich folgende Zeilen notwendig.

```
1  private var serverSocket:ServerSocket = new ServerSocket();
2  private var clientSocket:Socket;
3
4  public function SocketServer():void{
5    if (serverSocket.bound)
6    {
7    serverSocket.close();
8    serverSocket = new ServerSocket();
9    }
10   serverSocket.bind( localPort, localIP );
11   serverSocket.addEventListener( ServerSocketConnectEvent.CONNECT
     , onConnect );
12   serverSocket.listen();
13 }
```

Listing 5.1: ServerSocket

Nach dem Erzeugen eines neuen ServerSocket wird diesem ein Listener hinzugefügt, welcher darauf wartet, dass sich ein Client verbindet. Ist dies der Fall, so wird in der Funktion onConnect der Verweis auf den Client gespeichert und ihm neue Listener hinzugefügt. Der erste der beiden sorgt dafür, dass der Server auf Nachrichten vom Client reagieren kann. Der zweite informiert den Server darüber, wenn der Client die Verbindung getrennt hat.

```
1  if( clientSocket ) return;
2  clientSocket = event.socket;
```

```
clientSocket.addEventListener( ProgressEvent.SOCKET_DATA,
    onClientSocketData );
clientSocket.addEventListener( Event.CLOSE, onClientClosed );

sendCommand( AMConstants.COMMAND_CONNECT );
```

Auf Clientseite geht dieser Vorgang ähnlich einfach von Statten. Lediglich ein neuer Socket wird definiert und folgende Listener werden definiert.

```
socket = new Socket();

socket.addEventListener( Event.CLOSE, onSocketClose );
socket.addEventListener( Event.CONNECT, onSocketConnect );
socket.addEventListener( IOErrorEvent.IO_ERROR, onIOError );
socket.addEventListener( SecurityErrorEvent.SECURITY_ERROR,
    onSecurityError );
socket.addEventListener( ProgressEvent.SOCKET_DATA, onSocketData
    );

socket.connect( _address, _port );
```

Listing 5.2: Socket Client

Fast schon selbsterklärend informieren die Listener, falls eine Verbindung zustande kommt, sie abgebrochen wurde oder ein unerwarteter Fehler auftrat. Zum Schluss werden noch eingehende Daten weitergeleitet und verarbeitet.

5.2 Datenpaket

Alle Konstanten werden einheitlich in einer gemeinsam genutzten Klasse als static const gespeichert.

```
public class AMConstants{
public static const COMMAND_START_UPDATE:String =
                                    "COMMAND_START_UPDATE";
public static const COMMAND_FINISHED_UPDATE:String =
                                    "COMMAND_FINISHED_UPDATE";

public static const COMMAND_UPDATE_ENTITY:String =
                                    "COMMAND_UPDATE_ENTITY";
public static const COMMAND_UPDATE_SYSTEM:String =
                                    "COMMAND_UPDATE_SYSTEM";
...}
```

Listing 5.3: COMMANDS

Bei dem Versenden von Nachrichten ist die Übergabe eines Commandos unerlässlich. Für die grundlegenden NachrichtenTypen wird jeweils eine Funktion zur schnellen Ver-

wendung geschrieben.

```
1 public function sendString( command:String, string:String ):void{
2   var ba:ByteArray = new ByteArray();
3   ba.writeUTF(command);
4   ba.writeUTF(string);
5   socket.writeBytes( AMMessagePacker.packMessage( ba ));
6   socket.flush();
7 }
```

Listing 5.4: Nachrichten verschicken

Die Verbindung über Sockets verlangt ein ByteArray als Datenhalter. Aus diesem Grund wird jeder Paramter nacheinander in das Array geschrieben. Bevor es mit Hilfe von flush dann versendet wird, muss noch eine Längenangabe am Anfang des Arrays angefügt werden. Hierfür zuständig ist die Klasse AMMessagePacker. Sie liest das ByteArray ein und ermittelt dabei die Größe. All diese Informationen werden in einem neuen ByteArray gespeichert und als Rückgabewert geliefert.

Auf der gegenüberliegenden Seite, dem Debugger, trifft nun die Nachricht ein. Zunächst muss überprüft werden, ob die Nachricht bereits vollständig empfangen wurde. Es besteht die Möglichkeit, dass einzelne Bytes noch unterwegs sind. Hierzu wird ein Buffer angelegt, in dem alle empfangenen Daten gesammelt und erst bei Vollständigkeit weiter verarbeitet werden. Ist dies der Fall, so werden aus dem Buffer so viele Daten am Anfang ausgelesen und entfernt, wie der Längenoperator angibt. Die Daten können nun verarbeitet werden und der Buffer verkleinert sich wieder.

```
1 private function onClientSocketData( event:ProgressEvent ):void{
2 if (!clientSocket) return;
3
4 var tempBuffer:ByteArray = new ByteArray();
5 clientSocket.readBytes( buffer, buffer.length, clientSocket.
   bytesAvailable );
6
7 buffer.position = 0;
8 try{
9 while (bufferContentIsValid()){
10   var command:String = buffer.readUTF();
11   tempBuffer.clear();
12
13   switch (command)
14   {
15   case AMConstants.COMMAND_START_UPDATE:
16   [...]
17   break;
18   case AMConstants.COMMAND_FINISHED_UPDATE:
19   [...]
20   break;
21   [...]
22   default:
```

```
23    buffer.position = 0;
24    trace( "[GET] " + buffer.toString() );
25    buffer.clear();
26    break;
27  }
28
29  // clear used data from buffer
30  buffer.readBytes( tempBuffer, 0, buffer.bytesAvailable );
31  buffer.clear();
32  tempBuffer.readBytes( buffer );
33  }
34  }
35  catch ( error:Error )
36  {
37  trace("ERROR: " + error.getStackTrace());
38  }
39  }
```

Listing 5.5: Nachrichten empfangen

In der While-Schleife wird mit einem Switch-Case der Nachrichtentyp anhand des gesendeten Commands unterschieden. Entsprechend können die Daten weiterverarbeitet werden.

Dieser Vorgang ist auf beiden Seiten identisch.

5.3 Reflection

Das Analysieren von Klassen und der Auswertung der Eigenschaften kann sehr aufwendig werden. Dieses Thema an sich würde genügend Inhalt für eine weitere Bachelorarbeit liefern. Deshalb ist es empfehlenswert, auf eine vorgefertigte Library zurück zu greifen. In diesem Falle erfüllt die spicefactory alle nötigen Anforderungen.

Alleine mit einer Klasse, *ClassInfo*, lassen sich die meisten Probleme lösen. Diese analysiert ein Object und kann alle Eigenschaften, sowie dafür gesetzte Werte ausgeben. Zu beachten ist auch hierbei, dass nur mit public deklarierte Werte betroffen sind.

Mithilfe einer rekursiven Funktion werden alle Werte ausgelesen und in ein normales Object gespeichert. Besitzt dieser Wert keinen der Basistypen, so werden auch dessen Kindelemente auf die gleiche Art und Weiße analysiert. Dieser Vorgang ist relativ rechenintensiv und ist stark von der Anzahl der enthaltenen Werte und der Suchtiefe abhängig. Es ist dringend notwendig, die Suchtiefe zu begrenzen, da ansonsten Endlosschleifen bei ineinander verschachtelten Objekten möglich sind.

```
1  public static function analyzeObject( object:Object,
2  readable:Boolean = true,
```

```
3  writable:Boolean = true,
4  depth:int = 2 ):Object
5  {
6  if( depth < 0 || object == null ) return null;
7
8  var ci:ClassInfo = ClassInfo.forInstance( object );
9  var o:Object = new Object();
10 o.label = ci.name;
11 o.readable = readable;
12 o.writable = writable;
13
14 var children:Array = new Array();
15 var props:Array = ci.getProperties();
16 for( var i:int = 0; i < props.length; i++ ){
17   var prop:Property = props[i];
18   if( isBasicType( prop.type.getClass() )){
19     var basic:Object = new Object();
20     basic.label = prop.name;
21     if(prop.readable){
22       basic.value = prop.getValue( object );
23     }
24     else{
25       basic.value = null;
26     }
27
28     basic.readable = prop.readable;
29     basic.writable = prop.writable;
30     children.push( basic );
31   }
32   else{
33     var basic:Object = new Object();
34     basic.label = prop.name;
35     basic.readable = prop.readable;
36     basic.writable = prop.writable;
37     children.push( basic );
38     if (prop.readable){
39       var child:Object = analyzeObject( prop.getValue( object ),
40                                         prop.readable,
41                                         prop.writable,
42                                         depth - 1);
43
44       if (child != null){
45         children.push( child );
46       }
47     }
48   }
49 }
50 o.children = children;
51 return o;
52 }
```

Listing 5.6: Reflection

5.4 Monitoring

Flex kann nicht viel mit den serialisierten Daten anfangen. Sie müssen in einem speziellem Format vorliegen, damit später der Prozess des Bindings greift. Diese sogenannten ProxObjects überwachen alle ihre Eigenschaften und informieren die FlexKomponenten über eine Änderung, welche daraufhin den Inhalt aktualisieren.

Gespeichert werden alle Daten in einer ArrayCollection, welche später als DataProvider dient. DataProvider sind Sammlungen von Daten welche den Inhalt von Componenten beschreiben. In diesem Fall eine Art einfaches Array mit Objekten.

```
1  public static function updateSystem( system:Object ):void{
2  // check if system is already in the list
3  for( var i:int = 0; i < systemArray.length; i++ ){
4    if( systemArray[i].name == system.name ){
5      // system is already in the list, update all the values
6      systemArray[i].active = system.active;
7      systemArray[i].__updated = true;
8      return;
9    }
10 }
11 // system is new
12 system.__updated = true;
13 systemArray.push( system as ObjectProxy );
14 }
```

Listing 5.7: Monitoring auf Debugger-Seite

5.5 Benutzeroberfläche

Das Erstellen der Benutzeroberfläche in Flex ist ähnlich wie mit HTML. Die einzelnen Elemente bestehen aus Tags, welche weiter Eigenschaften enthalten können. Tags können beliebig tief geschachtelt werden, solange sie sich nicht überschneiden. Das bedeutet Tags welche innerhalb geöffnet wurden, müssen auch innerhalb geschlossen werden.

```
1  <mx:VDividedBox width="100%" height="100%">
2  <mx:HDividedBox width="100%" height="75%">
3
4  <!-- SystemList -->
5  <s:BorderContainer height="100%" width="20%">
6  <s:List id="SystemList" height="100%" width="100%"
7  itemRenderer="SystemItemRenderer"
8  dataProvider="{DataHandler.systemList}"
9  click="onSystemButtonClicked(event)"/>
10 </s:BorderContainer>
11
12 <!-- EntityList -->
```

```
13 <s:BorderContainer height="100%" width="20%">
14 <s:List id="EntiyList" height="100%" width="100%"
15 itemRenderer="EntityItemRenderer"
16 dataProvider="{DataHandler.entityList}"
17 click="clickEvent(event)"/>
18 </s:BorderContainer>
19
20 <!-- ComponentList -->
21 <s:BorderContainer height="100%" width="60%">
22 <mx:Tree id="tree" width="100%" height="100%"
23 dataProvider="{DataHandler.watchedEntity}"
24 itemRenderer="ComponentTreeItemRenderer"
25 showRoot="false"/>
26 </s:BorderContainer>
27 </mx:HDividedBox>
28 <mx:HDividedBox width="100%" height="25%">
29
30 <!-- StatsMonitor -->
31 <s:BorderContainer height="100%" width="90%">
32 <s:layout>
33 <s:HorizontalLayout horizontalAlign="center" verticalAlign="
    middle"/>
34 </s:layout>
35 <mx:LineChart id="chart" height="100%" width="80%"
36 dataProvider="{DataHandler.statsList}">
37 <mx:series>
38 <mx:LineSeries yField="MEMORY" form="segment"
39 displayName="memory usage in kb"
40 lineStroke="{s1}"/>
41 <mx:LineSeries yField="ENTITIES" form="segment"
42 displayName="active entities"
43 lineStroke="{s2}"/>
44 <mx:LineSeries yField="FPS" form="segment"
45 displayName="current frames per second"
46 lineStroke="{s3}"/>
47 </mx:series>
48 </mx:LineChart>
49 </s:BorderContainer>
50
51 <!-- Pause button -->
52 <s:Button id="pauseBtn"  height="100%" width="10%" click="{
    onPauseButtonClicked(event)}">
53 </s:Button>
54 </mx:HDividedBox></mx:VDividedBox>
```

Listing 5.8: Erstellung der Benutzeroberfläche

6 Anwendungsbeispiel 'Asteroids'

Um zu zeigen, wie einfach der entwickelte Debugger einzusetzen ist, wird er folgend an einem Beispiel eingesetzt. Hierbei handelt es sich um das Spiel Asteroids, programmiert von Richard Lord, welcher auch das Ash-Framework entwickelte.

6.1 Voraussetzung

Lediglich eine geeignete Entwicklungsumgebung und das AIR-SDK von Adobe sind nötig, um das Programm zu kompilieren. Das Spiel selbst benötigt nur eine externe Library, welche aber mit in der Ordnerstruktur zu finden ist.

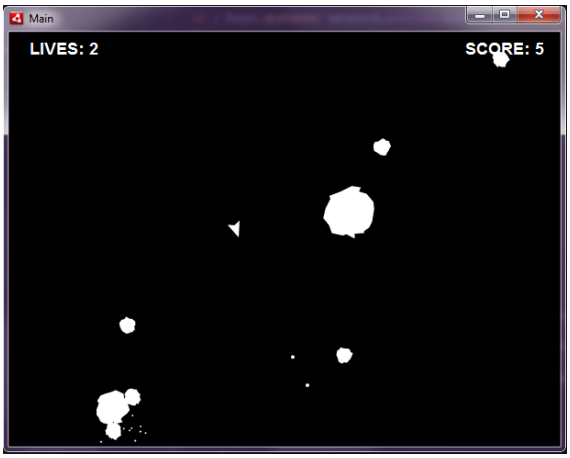

Abbildung 6.1: Asteroids - Portierung eines der ersten Computerspiele

6.2 Einbinden des Debuggers

Um den Debugger nutzen zu können, sind lediglich zwei Schritte notwendig. Der erste ist die Einbindung der AshMonitor-Library. War dies erfolgreich, reicht es aus, im Code folgende Zeile einzufügen:

```
new AshMonitor( engine );
```

An welcher Stelle dies geschieht, ist egal, jedoch muss zu dem Zeitpunkt der Erstellung Zugriff auf die *engine* bestehen, da ein Verweis auf diese als Pflichtparameter erwartet wird.

Weitere Parameter sind freiwillig und dienen nur dazu, in bestimmten Fällen Anpassungen vornehmen zu können. So etwa der Parameter der zu verwendenden IP, falls es zu

Konflikten kommt. Um zu überprüfen, ob alles richtig eingebunden wurde, muss lediglich das Spiel gestartet und ein Blick in die Konsole geworfen werden.

```
"C:\Program Files (x86)\AirSDK\AIRSDK_Compile
Player connected; session starting.
[trace] Start AshMonitor.
[trace] Try to connect...
[trace] No Debugger found at: 127.0.0.1:1500
```

Abbildung 6.2: Konsolenausgabe

6.3 Manipulation des Spieles

Wird nun vor oder während dem laufenden Spiel zusätzlich der Debugger gestartet, so sollten die beiden Programme sich direkt verbinden.

Sobald das Spiel läuft, zeigt der Debugger einige Informationen über das Spiel. Über den Pausebutton unten rechts kann ganz bequem das Spiel angehalten und fortgesetzt werden.

Durch das Deaktivieren des Kollisionsystems werden alle Kollisionen ausgeschaltet. Der Spieler kann nun nicht mehr von den Asteroiden getroffen werden. Jedoch reagieren diese auch nicht mehr auf die Schüsse des Spielers. Wird zusätzlich das BulletAgeSystem deaktiviert, so fliegen die abgefeuerten Schüsse unendlich über den Bildschirm. Die Entity-Anzahl erhöht sich stetig, wie im Statusmonitor zu sehen ist.

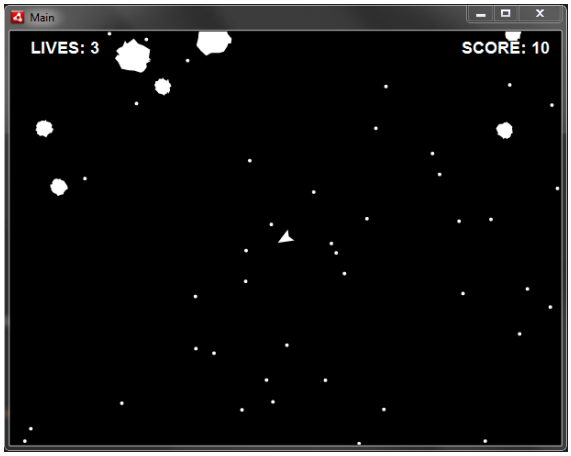

Abbildung 6.3: Asteroids mit deaktivierten Systemen

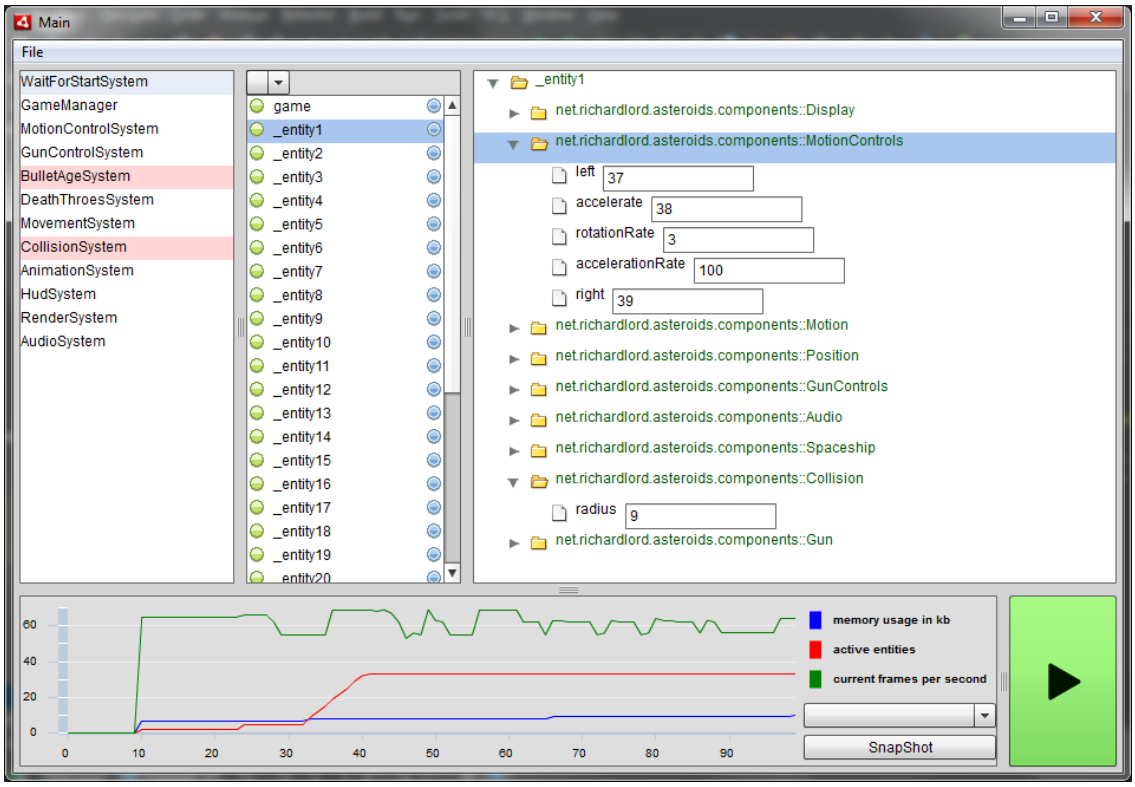

Abbildung 6.4: Debugger wärend das Spiel läuft

7 Schluss

Zusammengefasst wurde Folgendes erreicht: Ein externes eigenständiges Programm kann mit beliebigen Applikationen kommunizieren, welche die AshMonitor-Library benutzen. Es zeigt alle verwendeten Systeme und existierenden Entities. Der Nutzer hat die Möglichkeit, diese auch zu deaktivieren. Zudem können nähere Informationen über den Inhalt der Entities abgerufen werden. Zur Übersichtlichkeit kann die Applikation auch pausiert werden. Der Nutzer hat jederzeit eine Übersicht über den verwendeten Speicher und die aktuellen Frame-Per-Second.

Im letzten Kapitel wurde gezeigt, wie das Framework in vorhandene Projekte eingebunden wird und welche Möglichkeiten der Nutzer mit den vorhandenen Funktionen hat, um das Spiel zu beeinflussen.

7.1 Sicherheit

Das Programm und deren Verbindung zum Spiel kann ohne weiteres von Dritten abgehört und beeinflusst werden. Dies ist jedoch absolut kein Problem, da keine sensiblen Daten oder Quelltextinformationen versendet werden und auch kein dauerhafter Schaden durch Manipulation der Nachrichtenübertragung entsteht. Außerdem kann davon ausgegangen werden, dass das Programm vorrangig im privaten oder im abgesicherten Arbeitsumfeld genutzt wird.

7.2 Wiederverwendbarkeit

Auch wenn dieses Beispiel lediglich in ActionScript umgesetzt wurde, so kann dieser Debugger theoretisch auch von anderen Sprachen genutzt werden. Solange man im Stande ist, eine Verbindung über Sockets zu realisieren, müssen lediglich auf ClientSeite Anpassungen für die einzelnen Kommandos gemacht werden. Auch sollte der Datentyp String, welcher zur Übertragung dient, wohl in jeder anderen Sprache zur Verfügung stehen.

7.3 Ausblick

Das Thema eines Debuggers mit intuitiver Bedienoberfläche wird in den nächsten Jahren immer mehr an Bedeutung gewinnen. Das liegt zum einen an den immer größer werdenden Projekten mit einer Vielzahl an Arbeitskräften. Zum Anderen ist ein Trend in Richtung Entity-Component-Systemen festzustellen. Nehmen wir hier als Beispiel die

GameEngine Unity3D. Auch hier kann während der Laufzeit so ziemlich alles, mit Hilfe der hauseigenen in der Bedienoberfläche integrierten Schaltflächen, beeinflusst werden. Das reicht vom Erstellen neuer Objekte bis hin zum Ändern spezieller Eigenschaften.

Durch das Entfallen des Neustart des Programmes und des erneuten Compiliervorganges kann enorme Zeit in der Entwicklung gespart werden. Auch wenn dieser Vorgang nur eine halbe Minute benötigt, so wird dieser jedoch teilweise hunderte mal am Tag von den Programmieren durchgeführt.

Literaturverzeichnis

[1] James D. Murray, William vanRyper: *Graphics File Formats*. O'Reilly&Associates, Inc., Sebastopol, CA, 1994.

[2] ITWissen: *Debugger*.
http://www.itwissen.info/definition/lexikon/Debugger-debugger.html,
Stand: 30.09.2014

[3] Cory Janssen: *Adobe Flash*.
http://www.techopedia.com/definition/1991/adobe-flash,
Stand: 17.09.2014

[4] Joey Lott, Darron Schall, Keith Peters: *ActionScript 3.0 Cookbook*. O'Reilly Media, Inc., Sebastopol, CA, 2006, S. 409.

[5] Webhits: *Flash Player Verbreitung*.
http://www.webhits.de/deutsch/index.shtml?webstats.html,
Stand: 04.11.2014

[6] Richard Lord: *Ash entity framework*.
http://www.richardlord.net/blog/what-is-an-entity-framework,
Stand: 19.01.2012

[7] Deepa Subramaniam: *A brief overview of the Spark architecture and component set*.
http://www.adobe.com/devnet/flex/articles/flex4_sparkintro.html,
Stand: 08.03.2010

[8] Cory Janssen: *Action Script*.
http://www.techopedia.com/definition/1942/actionscript,
Stand: 17.09.2014

[9] ITWissen: *Serialisierung*.
http://www.itwissen.info/definition/lexikon/Serialisierung-serialization.html,
Stand: 07.10.2014

[10] Lee Brimelow: *Six reasons to use ActionScript 3.0*.
http://www.adobe.com/devnet/actionscript/articles/six_reasons_as3.html,
Erstellt: 18.08.2008

[11] ITWissen: *Transmission Control Protocol - TCP*.

http://www.itwissen.info/definition/lexikon/transmission-control-protocol-TCP-TCP-Protokoll.html,
Stand: 13.10.2014

[12] ITWissen: *Adobe Flex*.
http://www.itwissen.info/definition/lexikon/Adobe-Flex.html,
Stand: 23.09.2014

Listings

Erklärung

Hiermit erkläre ich, dass ich meine Arbeit selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt und die Arbeit noch nicht anderweitig für Prüfungszwecke vorgelegt habe.

Stellen, die wörtlich oder sinngemäß aus Quellen entnommen wurden, sind als solche kenntlich gemacht.

Mittweida, 28. Januar 2015